```
/*
    MSGBOX.PRG:  MsgBox(), a user interface building block function.
                 Calls should be made through the MESSAGE user-defined
                 command found in MSGBOX.CH.

        Author:  Craig Yellick
           Ver:  1.0a 27-Feb-91
*/

//  Required for using the MESSAGE command.
#include "msgbox.ch"

//  Required for using the ColorSet() function.
#include "colors.ch"

//  Other handy programming things.
#include "inkey.ch"
#command DEFAULT <a> := <b> => <a> := if(<a> = nil, <b>, <a>)


/*
 This #define will cause the testing/demo routine to be included in
  the compile. Delete the line and recompile to get rid of the test
  code and produce a smaller OBJ file.  However— leave the source
  code here so you can more easily test changes and additions in the
  future.
*/
#define TESTING
#ifdef TESTING
 function MsgTest()
 local dir_, sel, mess

   //  Must be called prior to using ColorSet().
   ColorInit()

   //  Clean up screen so this demo looks nice.
   setcursor(0)
   @ 0,0 clear

   message "This is a simple test of the MESSAGE command."
message "Can direct message to specific",;
        "row/col: 12,40." at 12,40

message "Multiple","Lines","Per","Message" at 1,50

message "Notice how box is being sized", ;
        "exactly large enough to hold the", ;
        "lines of text, automatically?" at 19, 1

message "Of course,","we can","control this", ;
        "by specifying","all the","coordinates." at 6,3 to 16,20

message "Press Any Key to Continue..." ;
         wait 0 at 8,30 color C_MESSAGE

@ 0,0 clear

message "We can also specify the WIDTH and DEPTH", ;
        "of the box directly, rather than trying to", ;
        "calculate the ending coordinates.  This", ;
        "box is directly set to be 10 by 40..." ;
         at 2,2 width 40 depth 10 color C_ENHANCED

//  Load an array with filenames
dir_ := {}
aeval(directory(), { |f_| aadd(dir_, f_[1]) } )

message "Better yet, we can specify that the lower", ;
         "part of the box be used as a scrolling", ;
         "window where you can make a selction.", "" , ;
         "Pick a file..." color C_ENHANCED ;
         choose dir_ choosecolor C_BOLD into sel ;
         at 8,32 depth 14

message "You selected: " +if(sel = 0, "<none>", dir_[sel]), ;
         "Press any key to continue..." color C_WARNING ;
         wait 0 at 9,38

//  Can stick message lines in an array.
```

```
    mess := {"There's always the option to restore the", ;
            "screen that was underneath, should we cover", ;
            "up something important.  Press a key to", ;
            "bring back the File Selection message."}

    message mess at 7,23 wait 0 color C_MESSAGE restore

    message "That's it for the demo..." at 18,1 ;
             color C_BLINK wait 5

    @ 0,0 clear

  return nil
#endif

/*---------------------------------*/


function MsgBox(r1, c1, r2, c2, width, depth, ;
                msg_, msgClr, ;
                ch_, chClr, ;
                wait, restore)
/*
   General-purpose message display and choice selection function.

   Parameter  Description
   ---------  -----------
   r1         Starting row, default is row().
   c1         Starting column, default is col().

   r2         Ending row, default based on messages/choices.
   c2         Ending column, default based on messages/choices.

   width      Columns wide, defaults to length of longest line.
   depth      Rows deep, defaults to number of message/choice lines.
              (Width and depth are ignored if r2 and/or c2
              are specified.

   msg_       Array of message lines.
   msgClr     Color number for main box and message section.
   ch_        Array of choice lines.
   chClr      Color number for choice section..

   wait       How long to wait (nil = don't)
   restore    True = restore screen afterwards, False/nil = leave.


                                Message Line 1
   General layout if            Message Line 2
   both Message lines           : etc
   and Choice lines
   are used:                    Choice Line 1
                                Choice Line 2
                                :  etc

 */

 local sel                    //  User's selection to return (if any)
 local msgLen := 0, chLen := 0 //  Length of longest line in msg_, ch_
 local boxWide, boxDeep       //  Box dimensions
 local oldCur, oldClr, oldScr //  Existing color, cursor and screen
 local oldR, oldC             //  Existing row/column position
 local crl                    //  Starting choice row
 local i                      //  Everyone's favorite iterator


   //  Some parameters have direct default values.
   default rl := row()
   default cl := col()
   default msg_ := {}
   default ch_  := {}
   default msgClr := C_NORMAL
   default chClr  := C_ENHANCED
   default restore := .f.

   //  Nil is used to indicate no message is desired, just choices.
   if (len(msg_) > 0) .and. (msg_[1] = nil)
     msg_ := {}
   endif
 /*
   Message and choices may be passed as a nested array due to the way
   the MESSAGE #command works, to provide maximum flexibility.
   Aclone() is used because we're dealing with an array reference, and
   yanking out a nested level may not please the calling routine which
   owns the array.
*/
if (len(msg_) > 0) .and. (valtype(msg_[1]) = "A")
  msg_ := aclone(msg_[1])
endif
if (len(ch_) > 0) .and. (valtype(ch_[1]) = "A")
  ch_ := aclone(ch_[1])
endif



 /*
   At this point we can determine some default dimensions. User can
    override them based on the "TO R, C" or DEPTH, WIDTH parameters.
  */
 //  Length of longest line in message area.
 aeval(msg_, { |s| msgLen := max(msgLen, len(s)) })

 //  Length of longest line in choice area.
 aeval(ch_,  { |s| chLen  := max(chLen,  len(s)) })

 //  Initial depth of box is messages plus choice lines.
 //  Add one line for bottom of box and one for separator line.
 boxDeep := if(len(msg_) = 0, 0, len(msg_) +1) ;
           +if(len(ch_)  = 0, 0, len(ch_)  +1)
 boxDeep := min(boxDeep, maxrow() -r1)

 //  Initial width of box is longest of message and choice lines.
 //  Add one column for right edge of box.
 boxWide := max(msgLen, chLen) +1
```

```
/*
   Some parameters can now be defaulted based on others.
*/
default r2 := r1 +if(depth = nil, boxDeep, depth)
default c2 := c1 +if(width = nil, boxWide, width)


/*
   Adjust confused parameters so programmer can see what's happened
  and keep on testing.
*/
if r2 <= r1
  r2 := maxrow()
endif
if c2 <= c1
  c2 := maxcol()
endif


/*
   Now that we know the dimensions we can save
   the screen region and draw the initial box.
*/
if restore
  oldScr := savescreen(r1, c1, r2, c2)
endif
oldR := row() ; oldC := col()
oldCur := setcursor(0)
oldClr := ColorSet(msgClr)
@ r1, c1 clear to r2, c2
@ r1, c1 to r2, c2


//  Display as many message lines as can fit in box.
//  Trim line widths to fit in interior of box.
for i := 1 to min(len(msg_), r2 -(r1 +1))
  devpos(r1 +i, c1 +1)
  devout(left(msg_[i], c2 -(c1 +1)))
next i
```

```
 //  If a choice array was sent.
 if len(ch_) > 0

   //  If a message array was sent.
   if len(msg_) >0
     //  Separator line between message and choice sections.
     devpos(r1 +len(msg_) +1, c1)
     devout(chr(195) +replicate(chr(196), c2-(c1+1)) +chr(180))
     cr1 := r1 +len(msg_) +2
   else
     cr1 := r1 +1
   endif

   //  Use "choices" color setting.
   ColorSet(chClr)

   //  ENTER or ESC key allow exit.
   do while .t.
     sel := achoice(cr1, c1 +1, r2 -1, c1 +(c2 -c1) -1, ch_)
     if (lastkey() = K_ENTER) .or. (lastkey() = K_ESC)
       exit
     endif
   enddo

 elseif valtype(wait) = "N"
   sel := inkey(wait)
 endif

 //  Restore the screen environment before exit.
 if restore
   restscreen(r1, c1, r2, c2, oldScr)
 endif
 setcolor(oldClr)
 setcursor(oldCur)
 devpos(oldR, oldC)

return sel

/* eof MsgBox.Prg */
```

```
/*
       File: MaxiBrow.Prg
     Author: Craig Yellick
       Date: 20-Apr-91
        Ver: 1.4a


  This is a general purpose database browser which attempts to use ALL
  of the TBrowse features in a single program. Yikes. The result is not
  intended to be a real routine that you'd implement in your
  applications. It goes to sometimes ridiculous lengths to incorporate
  TBrowse functions. It's intended to be a TBrowse playground of sorts
  where you can see various instance variables and methods in action
  and in the context of a large application rather than in small,
  isolated examples. Use it to experiment— make code changes, add new
  features, change constants, do whatever you find interesting. The
  more time you spend playing with TBrowse the better you'll understand
  it.

  The use of color and the formatting of the various status messages is
  deliberately spartan. I didn't want colorful and graphical things to
  get in the way of the underlying concepts. Plus, there's enough code
  as it is! The screen is by design very blah until you start fiddling
  around with highlighting and selection functions, or supply a
  database with dates, negative numbers, logical values or memo fields.
  Then you'll probably wish there weren't so many colors. I predict
  the incredibly flexible TBrowse color scheme will be responsible for
  a whole new generation of garish application screens. See the
  following comments for more details.

  All the features of this program are tied to various keystrokes. The
  source code comments surrounding the features describe the techniques
  being used. Briefly, here's what MaxiBrow can do:

  Navigation Keys
  Up, Down, Left, Right: Move one cell in any direction, will scroll
  rows up/down and pan columns left/right as needed.

  Home, End: Jump to first or last visible column.
```

```
Control-Home, Control-End: Jump to very first or very last column.

PageUp, PageDown: Scroll one screenful up or down.

Control-PageUp, Control-PageDown: Jump to very top or very bottom of
database.

Tab, Shift-Tab: Pan the screen left or right to see more columns.


Function Keys
=============


F1    Display record numbers/columns visited during "help". Each time
      you press F1 the current record number is added to a list. A
      column-based counter is also incremented. Press F1 periodically
      as you move through the database.  F1 also displays a brief
      summary of what most the other keystrokes do.

F2    Toggle between color and monochrome color schemes.

F3    Insert a copy of current column.

F4    Delete current column. Can't delete the last non-frozen column.

F5    Move window. Up/Down/Left/Right work as expected. Press any
      other key to finish the move. Built-in logic will prevent you
      from moving the window completely off the screen. Press
      backspace to restore coordinates to initial settings.

F6  Resize window. Up/Left make window larger, Down/Right make it
    smaller. Press any other key to finish the sizing. Built-in
    logic will prevent you from making window too small to be
    useful. Press backspace to restore coordinates to initial
    settings.

F7   Rotate column positions. Non-frozen columns are shifted to the
    left, first column is moved to the far right. Can press F7
    repeatedly to cycle through as many columns as you wish.  Press
    Shift-F7 to rotate columns in the other direction.
```

F8 Toggle drag-highlight navigation mode. After pressing F8 all cursor movements will enlarge a highlight box on the screen. Due to limitations inherent in TBrowse's colorRect() method the highlighted box is for the visible screen, only.

F9 Highlight current column. Due to aforementioned colorRect() limitation the highlight extends only for the rows currently visible.

F10 Highlight the current row. Entire row is highlighted, including columns that are not currently visible.

ESC Finished browsing, you'll be asked to confirm that you want to exit. Press Y to exit or any other key to remain in MaxiBrow.

Editing Keys

Alt-U	Toggle the global SET DELETED flag on/off.
Control-U	Toggle the individual record deletion flag on/off.
Enter	Edit current cell (including memo fields).
Control-Enter	Clear contents of current cell then edit (but not memo).
!..chr(255)	Edit current cell, start with current keystroke.

Other Misc Keys

Control-Left	Make current column more narrow. Can't make column smaller than one character wide.
Control-Right	Make current column more wide. Can't make column wider than width of original field in database.
Backspace	Clear all highlights and selections, reset all cargo instance variables, refresh entire screen. Used primarily to recover after making a mess.

Spacebar	Toggle record selection check-mark on and off. Counter at bottom of screen indicates how many have been selected. Unlike the F8, F9 and F10 highlighting functions, the check-marks are not limited to the currently visible screen. Backspace clears all selections.

Other Automatic Things

Other things will happen automatically depending on the contents of the database being browsed.

* Negative numbers will be in red.
* Logically false values will also be in red.
* All date columns will be in magenta.
* "Thud" sounds will be heard when you attempt to scroll or pan beyond the physical boundaries of the database.
* A relative position indicator, or "elevator", will be displayed on the left side of the window if the database contains more records than can fit in the window. Important note: The indicator is maintained independent of the index, if one is being used. This is implemented very elegantly thanks to the wonderful concept of TBrowse "skipBlocks".

Compiling and Running

```
Compile with: /n /w

Usage (from DOS): maxibrow datafile [indexfile]
```

```
________________________________

Comments
========

There's a tremendous amount of fancy browsing going on in here and to
help keep it all straight I've included lots of comments. Block-style
comments, like the ones you're reading here, are used to describe
large sections of code and also are used as headers to functions.
Single line comments are used when the comments are directed at the
next line (or small number of lines). Two blank lines separate
logical groups of source code, single blank lines separate small runs
of related source code within a major group.

Other helpful conventions:  If a function name is in all lowercase
letters then it's part of the built-in Clipper functions. If a
function name is in "proper" case it's a user-defined function and
can be found in this source code file. Array names end with a
trailing underscore. Manifest constants are in all upper case.

________________________________

File Contents
=============

Main(cFilename [, cIndexname])
  Main browsing program. From DOS, send database filename and option
  index filename.

Proper(cString)
  Given string, returns "properized" string where first character is
  made uppercase. Used in this program to make the database
  fieldnames look better.

YesNo(cMsg [, nSeconds])
  Given a message to display and optional maximum number of seconds
  to wait, return .t. if "Y" or "y" is pressed, .f. if not.

HelpStat(oBrowse)
  Given a browse object, use browse and current column cargo
```

variables to display stats about how often the F1-HELP key was pressed and which records were visited. (Not a general purpose help function, it's specific to this goofy program.)

FitInBox(nTop, nLeft, nBottom, nRight, aMessage)
Draw a box at specified coordinates and display array of message lines within the boundaries of the box, trim to fit if needed.

RecPosition([cHow] [, nHowmany])
This is a single function used for all three TBrowse data positioning blocks. Given the type of movement required (top, bottom or skip) and how many records to skip, function performs all the necessary database movements. If called without any parameters the function returns the current record's position within the database. The position is maintained independent of the index, if any.

RecDisplay(nRec, aList, lDeleted)
This function is used by the record-# column data retrieval block to format the record number for display. It handles the check-mark. The current record number is passed along with an array containing record numbers that should be displayed with check-marks. Records marked for deletion have an asterisk in the column.

aCount(aX, bCountBlock [, nStart] [, nCount])
Given an array, use the supplied code block to count the number of elements that match a condition. Optional starting element number and number of elements to evaluate. Returns count.

aInsert(aX, nPos [, xValue])
Increase size of array by inserting new value in specified position. This function combines the effects of the ains() and asize() functions.

aDelete(aX, nPos)
Decrease size of array by deleting element in specified position. This function combines the effects of the adel() and asize() functions.

aRotate(aX [,lDir])

```
    Rotates elements in array passed as parameter. Elements are shifted
    up one. First element is shifted to last element. Optional
    direction to shift, default is .t. and implies shift up, .f. is
    down. Returns nil.

  ColumnColor(xValue)
    Given a value of any type, returns a color-selection array based on
    the type and value. Used to install logic for displaying certain
    data types or values in special colors. For example, negative
    numbers in red. Any number of cases can be installed.

  Navigate(objBrowse, nKey)
    Given a browse object and a potential navigation key, function
    searches its internal list of keystrokes and associated browse
    navigation methods. If key is found the method is sent to the
    browse and the function returns .t., if not found, function
    returns .f. and no action is taken.

  EditCell(objBrowse, cFieldName, cColor)
    A general-purpose browse cell contents editor, works with all
    database field types including memo fields. All editing, including
    memo-edit, occurs within the browse window regardless of its
    current size or position. On exit from cell the function passes
    along browse navigation when appropriate.

  ________________________________________
*/


/*
  Establish some helpful preprocessor directives.
*/
#include "INKEY.CH"
#include "DBSTRUCT.CH"

#define K_SPACE      32
#define K_CTRL_ENTER 10

#define THUD      tone(60, 0.5)
#define BADKEY    tone(480, 0.25); tone(240, 0.25)
```

```
#define lstr(n)  ltrim(str(n))

#define INIT_R1   4
#define INIT_R2  (maxrow() -4)
#define INIT_C1  10
#define INIT_C2  (maxcol() -10)

#define FREEZE_COL  1


/*
   Default color scheme for all columns.
   (Used with instance variable browse:colorSpec.)

   1: Regular cell
   2: Highlighted regular cell
   3: Block-selection cell
   4: Highlighted block-selection cell
   5: Checked record-#
   6: Highlighted, checked record-#
   7: Regular negative numbers and .F. values
   8: Highlighted negative numbers and .F. values
   9: Regular dates
  10: Highlighted dates

                    1    2    3     4     5    6     7     8      9    10
*/
#define COL_COLOR "W/N, N/W, W+/B, B/W, W+/G, B+/G, R+/N, W+/R, RB+/N, W+ RB"
#define COL_MONO  "W/N, N/W, N/W,  W*/N, W/N, W+/N, W+/N, N/W,  W+/N,  N/W"

/*
  The following make it easier to use the browse:colorSpec.
  They correspond to the color scheme defined above.
*/
#define REGULAR_CELL  {1,2}
#define BLOCKED_CELL  {3,4}
#define CHECKED_CELL  {5,6}
#define NEGVAL_CELL   {7,8}
#define DATE_CELL     {9,10}
```

```
//  This next one is for the GET/READ feature,
//  defined here for consistency with rest of browse.
#define EDIT_COLOR     "W+/G"

/*———————————————————————————————*/

function Main(filename, indexname)
/*
   Main browsing function.
*/

local r1, r2, c1, c2, scr, fileDescr
local column, browse, key
local stru_, recs_
local s, n, w
local hiRow, hiCol, hiRow2, hiCol2
local dragMode := .f., delSwitch := .f.
local temp, useColor, relPos


  //  Check that command line parameters are kosher.
  if filename = nil
    ? "Must specify a database filename and optionally an index filename."
    quit
  elseif .not. (file(filename) .or. file(filename +".DBF"))
    ? "Database file does not exist."
    quit
  endif


  //  Get rid of the cursor and start with a clean slate.
  setcursor(0)
  @ 0,0 clear
  set scoreboard off


  //  Open the database and index.
  use (filename) new
  fileDescr := "File: " +upper(filename)
```

```
if (indexname <> nil) .and. (file(indexname) ;
                        .or. file(indexname +".NTX"))
  set index to (indexname)
  fileDescr += ", Index: " +upper(indexname)
endif


// Assign initial browse window coordinates.
r1 := INIT_R1
r2 := INIT_R2
c1 := INIT_C1
c2 := INIT_C2
@ r1 -2, c1 +((c2 -(c1 +len(fileDescr))) /2) say fileDescr
@ r1 -1, c1 -1, r2 +1, c2 +1 box " █▀███▄██ "


// Create a new browse object.
browse := TBrowseNew()


/*
   Things that affect the entire browse.
*/
// Assign window coordinates.
browse:nTop := r1
browse:nBottom := r2
browse:nLeft := c1
browse:nRight := c2
// Assign heading, footing and column separators.
browse:headSep := " ─┬─ "
browse:colSep  := " │ "
browse:footSep := " ═╧═ "
// Cargo will be used later, associated with the F1 key.
browse:cargo := {}


// Assign default color scheme according to adapter card.
useColor := iscolor()
browse:colorSpec := if(useColor, COL_COLOR, COL_MONO)
```

```
//  All three position blocks get routed through a single function.
//  This allows us to do some amazing things, later.
browse:goTopBlock    := { | | RecPosition("top")     }
browse:goBottomBlock := { | | RecPosition("bottom")  }
browse:skipBlock     := { |n| RecPosition("skip", n) }


/*
   First column will be the record number.
   We're going to do some tricky things with this column
   so setting it up is more complex than normally necessary.
*/
//  This array will keep track of the records visited each
//  time the F1-HELP key is pressed.
recs_ := {}

//  Create a new column object.
column := TBColumnNew()

//  The RecDisplay() function provides the check-mark toggle
//  that's associated with the spacebar key.
column:block := { || RecDisplay(recno(), recs_, deleted()) }

//  The footing line will be used to display field type and width.
column:heading := " Rec-#"
column:footing := "  Type:; Col-#:"

//  We want this column to have a different color when "checked".
column:colorBlock := { |r| if("*" $ r, CHECKED_CELL, REGULAR_CELL) }

//  Column cargo is used later store a count of how many times
//  the F1-HELP key was pressed in each column.
column:cargo := 0

//  Add the record-# column just defined to the main browse object.
browse:addColumn(column)


/*
```

```
     The remainder of the columns in the browse will be comprised
     of the fields in the current database.
  */

  //  For each field in the database...
  //    (See documentation for the dbstruct() function and
  //    details about what the stru_ array contains.)
  stru_ := dbstruct()
  for n := 1 to len(stru_)

    //  Create a column object for each field.
    column := TBColumnNew()

    //  Heading is the field name, footing is the type and width.
    //  For example, a 12 character field would be "C:12".
    //  Columns are numbered in a second line in the footing, (n).
    column:heading := Proper(lower(stru_[n, DBS_NAME]))
    column:footing := stru_[n, DBS_TYPE] +":" +lstr(stru_[n, DBS_LEN]) ;
                      +";(" +lstr(n) +")"

    //  Date-type columns get special color scheme.
    if stru_[n, DBS_TYPE] = "D"
      column:defColor := DATE_CELL
    else
      //  Make some of the colors depend on cell value.
      column:colorBlock := { |v| ColumnColor(v) }
    endif


    //  Data-retrieval blocks based simply on the field value.
    //  Don't create a block for memo fields.
    if stru_[n, DBS_TYPE] <> "M"
      column:block := fieldblock(stru_[n, DBS_NAME])
      column:width := stru_[n, DBS_LEN]
    else
      column:block := { || " memo " }
      column:width := 6
    endif
    //  Initialize cargo, we'll be using it later.
    column:cargo := 0


    //  First column after frozen one (in this case, the
    //  record-#) gets a different set of separators to
    //  better divide "frozen" columns from the scrollable ones.
    //
    if n = FREEZE_COL
      column:headSep := "═╦═"
      column:colSep  := " ║ "
      column:footSep := "═╩═"
    endif

    //  Add the new column object to the main browse object.
    browse:addColumn(column)
  next n


  //  Freeze the first column (the record-#).
  browse:freeze := FREEZE_COL


  //  Move cell pointer beyond frozen column(s).
  browse:colPos := browse:freeze +1


  //  We'll handle our own highlighting, thank you.
  browse:autoLite := .f.


  //  Used later to mark relative pointer position on left edge of window.
  relPos := 1


  /*
     Finally!  We're done getting everything set up.
     Allow user to play with the browse until exit is confirmed.
  */
  do while .t.
```

```
//  Can't move beyond last column.
//  This condition will be fixed up by stabilize(),
//  so we must check for it prior to stabilization.
if browse:colPos > browse:colCount
  THUD
endif


//  Can't move into frozen column.
if browse:colPos <= browse:freeze
  THUD
  browse:colPos := browse:freeze +1
endif


//  Stabilize the display, if it needs to be. Use of the nextkey()
//  function allows us to exit the loop if a keystroke occurs, but
//  without disturbing the contents of the keyboard buffer.
if .not. browse:stable
  @ 0,0 say "STABILIZING..."
  do while .not. browse:stabilize()
    if nextkey() <> 0
      exit
    endif
  enddo
  @ 0,0
endif


//  These get updated during the stabilize,
//  so they can't be checked until after stabilize finishes.
if browse:hitTop .or. browse:hitBottom
  THUD
endif


//  If in "drag the highlight around" mode, update
//  the rectangle coordinates and display it.
if dragMode
  hiRow  := min(hiRow,  browse:rowPos)
  hiCol  := min(hiCol,  browse:colPos)
  hiRow2 := max(hiRow2, browse:rowPos)
  hiCol2 := max(hiCol2, browse:colPos)
  browse:colorRect({hiRow, hiCol, hiRow2, hiCol2}, BLOCKED_CELL)
endif


//  Update relative position indicator, but only if
//  there are more records in database than can fit on the screen.
if lastrec() > browse:rowCount
  @ browse:nTop +2 +relPos, browse:nLeft -1 say "█"
  relPos := min((RecPosition()/lastrec()) *browse:rowCount, ;
                browse:rowCount -1)
  @ browse:nTop +2 +relPos, browse:nLeft -1 say chr(18) color "I"
endif

//  Update the "more columns left" and "more columns right" indicators.
//  Start by clearing existing indicator arrows, if any.
@ browse:nTop, browse:nLeft  -1 say " " color "I"
@ browse:nTop, browse:nRight +1 say " " color "I"
if browse:leftVisible > (browse:freeze +1)
  @ browse:nTop, browse:nLeft  -1 say chr(27) color "I"
endif
if browse:rightVisible < browse:colCount
  @ browse:nTop, browse:nRight +1 say chr(26) color "I"
endif


/*
   The bottom three rows of the screen are used to display status
   information about various pieces of the browse and column
   objects. Watch these lines as you navigate in the database.
*/

//  Display info about the browse window.
@ maxrow() -2, 0
?? "Browse: Row " +lstr(browse:rowPos)
?? ", Col " +lstr(browse:colPos)

@ maxrow() -1, 0
```

```
?? "Absolute DBF position: " +lstr(RecPosition())
?? "  (" +lstr( round((RecPosition()/lastrec()) *100, 0)) +"%)"

@ maxrow(), 0
column := browse:getColumn(browse:colPos)
?? "Record " +lstr(recno()) +": " +column:heading +" = "
//
//  Use of @..SAY will allow long strings to display off
//  the edge of the screen, rather than wrapping around.
//
@ row(), col() say eval(column:block)


s := "[ F1:HELP ]"
@ maxrow() -2, (maxcol() -len(s)) /2 say s


s := "Records -Marked: " +lstr(aCount(recs_, { | e | (e <> nil) }) )
@ maxrow() -2, maxcol() -len(s) say s
s := "LastKey = " +lstr(lastkey())
@ maxrow() -1, maxcol() -len(s) say s
s := "NextKey = " +lstr(nextkey())
@ maxrow(), maxcol() -len(s) say s


//  Highlight cell pointer and wait for keystroke.
browse:hilite()
key := inkey(0)

/*
   Take action on the keystroke. Could be cursor navigation
   or any of a large number of browse-modification features.
*/
do case


//  If the general browse navigation function returns .t.
//  it means it handled the key for us.
//
case Navigate(browse, key)
case key = K_CTRL_LEFT  //  Decrease column width (if we can).
  //
  //  stru_[colPos -1] because first column is record number.
  //
  w := browse:getcolumn(browse:colPos):width
  if w > 1
    browse:getcolumn(browse:colPos):width--
    //  Update the footing to reflect the new width.
    browse:getcolumn(browse:colPos):footing ;
     := stru_[browse:colPos -1, DBS_TYPE] +":" +lstr(--w) ;
     +";(" +lstr(browse:colPos) +")"
    browse:configure()
  else
    THUD
  endif


case key = K_CTRL_RIGHT  //  Increase column width (if we can).
  //
  //  stru_[colPos -1] because first column is record number.
  //
  w := browse:getcolumn(browse:colPos):width
  if w < stru_[browse:colPos -1, DBS_LEN]
    browse:getcolumn(browse:colPos):width++
    //  Update the footing to reflect the new width.
    browse:getcolumn(browse:colPos):footing ;
     := stru_[browse:colPos -1, DBS_TYPE] +":" +lstr(++w) ;
     +";(" +lstr(browse:colPos) +")"
    browse:configure()
  else
    THUD
  endif


case key = K_F1          //  Display help/cargo status.
  //
  HelpStat(browse)
```

```
case key = K_F2          //  Toggle colorSpec between color and mono.
  //
  useColor := .not. useColor
  browse:colorSpec := if(useColor, COL_COLOR, COL_MONO)
  browse:configure()


case key = K_F3          //  Insert copy of current column.
  //
  //  stru_[colPos -1] because first column is record number.
  //
  //  Must adjust the stru_ array so it stays accurate.
  //
  aInsert(stru_, browse:colPos -1, stru_[browse:colPos -1])
  browse:insColumn(browse:colPos, browse:getColumn(browse:colPos))


case key = K_F4          //  Delete current column.
  //
  //  stru_[colPos -1] because first column is record number.
  //
  //  Don't allow deletion of last non-frozen column.
  //  Must adjust the stru_ array so it stays accurate.
  //
  if browse:colCount > (browse:freeze +1)
    aDelete(stru_, browse:colPos -1)
    browse:delColumn(browse:colPos)
  else
    THUD
  endif


case key = K_F5          //  Move the window.
  //
  //  Don't allow window to be pushed completely off the screen,
  //  force atleast a few rows and columns to say visible, TBrowse
  //  is capable of hanging the computer under certain oddball
  //  situations.
  //
```

```
scr := savescreen(0,0,maxRow(),maxCol())
@ 0,0
@ 0,0 say "Move window: " +chr(18) +" " +chr(29)
do while .t.
  @ r1 -1, c1 -1, r2 +1, c2 +1 box replicate("·", 8)
  key := inkey(0)
  restscreen(0,0,maxRow(),maxCol(), scr)
  do case
  case key = K_UP
    if r2 > 4
      r1–
      r2–
    else ; THUD; endif
  case key = K_DOWN
    if r1 < (maxRow() -4)
      r1++
      r2++
    else ; THUD; endif
  case key = K_LEFT
    if c2 > 10
      c1–
      c2–
    else ; THUD; endif
  case key = K_RIGHT
    if c1 < (maxCol() -10)
      c1++
      c2++
    else ; THUD; endif
  case key = K_BS  // Restore initial values
    r1 := INIT_R1
    r2 := INIT_R2
    c1 := INIT_C1
    c2 := INIT_C2
  otherwise
    exit
  endcase
enddo
restscreen(0,0,maxRow(),maxCol(), scr)
@ browse:nTop -2, browse:nLeft -1 ;
  clear to browse:nBottom +1, browse:nRight +1
```

```
       @ r1 -2, c1 +((c2 -(c1 +len(fileDescr))) /2) say fileDescr
       @ r1 -1, c1 -1, r2 +1, c2 +1 box "█▀▀▀▙▀█"
       browse:nTop := r1
       browse:nBottom := r2
       browse:nLeft := c1
       browse:nRight := c2


     case key = K_F6           //   Resize the window.
       //
       //  Don't allow resize unless entire window is visible,
       //  TBrowse might hang the computer if things get too wierd.
       //  Also, don't let size get too small or too large.
       //
       if (r1 < 0) .or. (c1 < 0) ;
          .or. (r2 > maxRow()) .or. (c2 > maxCol())
          BADKEY
       else
         scr := savescreen(0,0,maxRow(),maxCol())
         @ 0,0
         @ 0,0 say "Resize window: " +chr(18) +" " +chr(29)
         do while .t.
           @ r1 -1, c1 -1, r2 +1, c2 +1 box replicate("·", 8)
           key := inkey(0)
           restscreen(0,0,maxRow(),maxCol(), scr)
           do case
           case key = K_UP
             if (r2 -r1) < (maxRow() -1)
               r1--
               r2++
             else ; THUD; endif
           case key = K_DOWN
             if (r2 -r1) > 4
               r1++
               r2--
             else ; THUD; endif
           case key = K_LEFT
             if (c2 -c1) < (maxCol() -3)
               c1--
               c2++
        else ; THUD; endif
      case key = K_RIGHT
        if (c2 -c1) > 8
          c1++
          c2-
        else ; THUD; endif
      case key = K_BS  // Restore initial values
        r1 := INIT_R1
        r2 := INIT_R2
        c1 := INIT_C1
        c2 := INIT_C2
      otherwise
        exit
      endcase
    enddo
    restscreen(0,0,maxRow(),maxCol(), scr)
    @ browse:nTop -2, browse:nLeft -1 ;
      clear to browse:nBottom +1, browse:nRight +1
    @ r1 -2, c1 +((c2 -(c1 +len(fileDescr))) /2) say fileDescr
    @ r1 -1, c1 -1, r2 +1, c2 +1 box "█▀█▄█"
    browse:nTop := r1
    browse:nBottom := r2
    browse:nLeft := c1
    browse:nRight := c2
  endif


case key = K_F7 .or. ;   //  Rotate non-frozen column positions +/-.
     key = K_SH_F7
  //
  @ 0,0 say "ROTATING COLUMNS..."
  if key = K_F7
    temp := browse:getColumn(browse:freeze +1)
    for n := (browse:freeze +1) to (browse:colCount -1)
      browse:setcolumn(n, browse:getColumn(n +1))
    next n
    browse:setcolumn(browse:colCount, temp)
  else
    temp := browse:getcolumn(browse:colCount)
    for n := browse:colCount to (browse:freeze +2) step -1
```

```
          browse:setcolumn(n, browse:getcolumn(n -1))
        next n
        browse:setcolumn(browse:freeze +1, temp)
      endif
      //
      //  Also rotate database structure array so
      //  anything that depends on it remains accurate.
      //
      aRotate(stru_, key == K_F7)
      @ 0,0


    case key = K_F8           //  Drag-highlight mode.
      //
      //  Initialize only if not already in drag-highlight mode.
      //
      if .not. dragMode
        hiRow := hiRow2 := browse:rowPos
        hiCol := hiCol2 := browse:colPos
      endif
      dragMode := .not. dragMode


    case key = K_F9           //  Highlight current column.
      //
      browse:colorRect({1, browse:colPos, ;
                        browse:rowCount, browse:colPos}, ;
                        BLOCKED_CELL)

      //  Move over one column, a convenience feature.
      if browse:colPos > browse:colCount
        *  Wrap to first column?
      else
        browse:right()
      endif


    case key = K_F10          //  Highlight current row.
      //
      browse:colorRect({browse:rowPos, browse:freeze +1, ;
                         browse:rowPos, browse:colCount}, ;
                         {3,4})

    //  Move down one row, a convenience feature.
    if browse:hitBottom
      *  Wrap to top?
    else
      browse:down()
    endif


  case key = K_BS        //  Clear, zero and refresh everything in sight.
    //
    //  stru_[n -1] because first column is record number.
    //
    @ 0,0 say "CLEANING UP..."
    dragMode := .f.
    recs_ := {}
    for n := (browse:freeze +1) to browse:colCount
      browse:getcolumn(n):cargo := 0
      browse:getcolumn(n):width := stru_[n -1, DBS_LEN]
      browse:getcolumn(n):footing := stru_[n -1, DBS_TYPE] ;
                           +":" +lstr(stru_[n -1, DBS_LEN]) ;
                           +";(" +lstr(n) +")"
    next n
    browse:cargo := {}
    browse:configure()
    @ 0,0


  case key = K_SPACE    //  Toggle record marker on/off.
    //
    n := ascan(recs_, recno())
    if n = 0
      n := ascan(recs_, nil)
      if n = 0
        aadd(recs_, recno())
      else
        recs_[n] := recno()
      endif
      else
        adel(recs_, n)
      endif

      //  Force this row to be refreshed. If user marked it
      //  we want to be certain they're seeing the most up-to-date data.
      browse:refreshCurrent()

      //  Move down to next row as a convenience for user.
      browse:down()


    case key = K_ALT_U    //  Toggle SET DELETED on/off.
      //
      if (delSwitch := .not. delSwitch)
        set deleted on
      else
        set deleted off
      endif
      browse:refreshAll()


    case key = K_CTRL_U   //  Toggle the record deletion flag.
      //
      if deleted()
        recall
      else
        delete
      endif
      browse:refreshCurrent()


    case (key = K_ENTER) ;        //  Open current cell for editing.
    .or. (key = K_CTRL_ENTER) ;   //  Clear cell contents and edit.
    .or. (key > K_SPACE)          //  Edit by starting to type.
      //
      EditCell(browse, ;
               stru_[browse:colPos -1, DBS_NAME], ;  //  Field name
               EDIT_COLOR)
    case key = K_ESC      //  Done browsing.
      //
      //  Turn off hilite, user's attention should be at y/n prompt.
      //
      browse:deHilite()
      if YesNo("Exit? Are you sure?")
        exit
      endif


    //  Undefined key, be-boop to let user
    //  know that we heard but can't obey.
    //
    otherwise
      BADKEY
    endcase

  enddo  //  While browsing.

  setcursor(1)
  @ maxrow(), 0

return nil


/*-----------------------------------*/


function Proper(s)
/*
   Return "properized" version of string, first letter made uppercase.
   Used in column headings to make the field names look more nice.
*/
return upper(left(s, 1)) +lower(substr(s, 2))


/*-----------------------------------*/
```

```
function YesNo(msg, time)
/*
    Display yes/no question message in box centered on screen, wait up
    to so many seconds before assuming "no". This function takes pains
    not to disturb the calling routine's screen/color/cursor settings.
*/
local k, scr, curs, clr
  scr := savescreen(11,0,13,maxCol())
  msg := " " +msg +" "
  curs := setcursor(0)
  clr := setcolor( if(iscolor(), "GR+/R", "W+*/N") )
  @ 11, (maxCol()/2) -(len(msg)/2) -1 ;
    to 13, (maxCol()/2) +(len(msg) /2) double
  @ 12, (maxCol()/2) -(len(msg)/2) say msg
  k := inkey(if(time = nil, 0, time))
  restscreen(11,0,13,maxCol(), scr)
  setcolor(clr)
  setcursor(curs)
return (chr(k) $ "Yy")


/*————————————————————————————————————*/


function HelpStat(b)
/*
   Display help and status screen. You can do pretty well anything you
   want for "help". In this case we're displaying some interesting
   stats about where the cell pointer was sitting when help was pressed.
*/
local clr, scr := savescreen(0, 0, maxrow(), maxcol())

  //  Look for current record number in the browse cargo,
  //  add it to list of records if not found.
  if ascan(b:cargo, recno()) = 0
    aadd(b:cargo, recno())
  endif

  @ 0, 0 clear
  @ 0, 0 to 4, maxCol()
@ 1, 2 say "Browse and Column Cargo..."

//  Display list of record numbers maintained in browse cargo.
@ 2, 2 say "  Record-#s visited when HELP was pressed:"
aeval(b:cargo, { |rec| qqout(" " +lstr(rec)) } )

//  Display current column cargo count, then increment it.
@ 3, 2 say "  Prior times HELP pressed in this column: " ;
           +lstr(b:getColumn(b:colPos):cargo++)

FitInBox(5, 0, 16, 35, ;
        {"         Navigation Keys        ", "", ;
         "Up·Dn·Lt·Rt         Take a guess", ;
         "Home·End       First/last column", ;
         "^Home·^End   Very first/last col", ;
         "PgUp·PgDn            See up/down", ;
         "^PgUp·^PgDn    First/last record", ;
         "Tab·Shf-Tab  Pan cols left/right", "", ;
         "           ESC Exits            "})

FitInBox(maxrow() -19, maxcol() -42, maxrow(), maxcol(), ;
        {"F2          Toggle between color/mono", ;
         "F3          Insert copy of current column", ;
         "F4          Delete current column", ;
         "F5          Move window (BS=reset)", ;
         "F6          Resize window (BS=reset)", ;
         "F7          Rotate column positions (Shift-F7)", ;
         "F8          Toggle drag-highlight on/off", ;
         "F9          Highlight current column", ;
         "F10         Highlight current row", "", ;
         "Alt-U       Toggle SET DELETED on/off", ;
         "^U          Toggle record delete on/off", ;
         "Enter       Edit current cell (incl memo)", ;
         "^Enter      Clear cell then edit", ;
         "^Left       Make column more narrow", ;
         "^Right      Make column more wide", ;
         "Spacebar    Toggle ·-record", ;
         "Backspace   Clear and reset everything"})
```

```
 @ maxrow() -4, 0 say "See Detailed Comments in Source Code"
 clr := setcolor("I")
 @ maxrow() -3, 0 say replicate("█", 36)
 @ maxrow() -2, 0 say "     MaxiBrow by Craig Yellick      "
 @ maxrow() -1, 0 say "     Ver 1.4a        20-Apr-91      "
 @ maxrow(),    0 say replicate("█", 36)
 setcolor(clr)

  inkey(0)
  restscreen(0, 0, maxrow(), maxcol(), scr)

return nil


/*————————————————————————————————————*/

function FitInBox(r1, c1, r2, c2, msg_)
/*
   Draw a box of specified dimensions and display the contents
   of an array of message lines in it. Display only what will
   fit within the box boundaries.
*/
local i

  @ r1, c1 clear to r2, c2
  @ r1, c1 to r2, c2 double
  for i := 1 to min(len(msg_), r2 -r1 -1)
    @ r1 +i, c1 +2 say left(msg_[i], c2 -c1 -1)
  next i

return nil

/*————————————————————————————————————*/


function RecPosition(how, howMany)
/*
   General-purpose record positioning function, called by TBrowse goTop,
   goBottom and skip blocks. Returns number of record actually moved if
   in "skip" mode.
   Also can be called with no parameters to get record position within
   database independent of presence of index.
*/

//  Assume no movement was possible
local actual := 0

local i
static where := 1

  do case
  case how = "top"
    where := 1
    goto top

  case how = "bottom"
    where := lastrec()
    goto bottom

  case how = "skip"
    do case
    //  Moving backwards
    case howMany < 0
      do while (actual > howMany) .and. (.not. bof())
        skip -1
        if .not. bof()
          actual--
        endif
      enddo

    //  Moving forwards
    case howMany > 0
      do while (actual < howMany) .and. (.not. eof())
        skip +1
        if .not. eof()
          actual++
        endif
      enddo
      if eof()
```

```
          skip -1
        endif

      //  No movement requested, re-read current record
      otherwise
        skip 0
      endcase

    //  No parameters passed, return current position.
    otherwise
      return where
    endcase

    //  Update position tracker and prevent boundary wrap.
    where += actual
    where := min(max(where, 1), lastrec())

  return actual


  /*————————————————————————————————*/


  function RecDisplay(rec, list_, del)
  /*
    Returns specified record number plus indicator if record has been
    placed in list_ array. Intended for use in TBColumn retrieval block.
  */
  return if(del, " *","  ") +str(rec,4) ;
        +if(ascan(list_, rec) = 0, "  ", " •")


  /*————————————————————————————————*/


  function aCount(a_, countBlock, start, count)
  /*
    Given array and code block, return number of elements that evaluate
    true.
  */
```

```
local howMany := 0
  aeval(a_, ;
    { |elem| howMany += if(eval(countBlock, elem), 1, 0) }, ;
    start, count)
return howMany


/*————————————————————————————————————————*/


function aInsert(a_, pos, value)
/*
   Increase size of array by inserting new value in specified position.
*/
  asize(a_, len(a_) +1)
  ains(a_, pos)
  a_[pos] := value
return nil


/*————————————————————————————————————————*/


function aDelete(a_, pos)
/*
   Decrease size of array by removing element at specified position.
*/
  adel(a_, pos)
  asize(a_, len(a_) -1)
return nil


/*————————————————————————————————————————*/


function aRotate(a_, up)
/*
   Rotate array elements such that first is last, last is first, and all
   others shift up one position. If UP is passed and is false, the shift
   direction is reversed.
```

```
*/
local temp
  if (up = nil) .or. up
    temp := a_[1]
    aeval(a_, { |e,n| a_[n] := a_[n +1] }, 1, len(a_) -1)
    a_[len(a_)] := temp
  else
    //
    //  Yes, it's possible to traverse an array backwards with aeval()!
    //
    temp := a_[len(a_)]
    aeval(a_, { |e,n| a_[len(a_) -(n-1)] := a_[len(a_) -n] }, ;
               1, len(a_) -1)
    a_[1] := temp
  endif
return nil


/*——————————————————————————————————*/


function ColumnColor(value)
/*
   Color selection used in TBColumn colorBlock. Allows each data type to
   have it's own color scheme.
*/
local type, clr
  type := valtype(value)
  do case
  case (type = "N") .and. (value < 0)
    clr := NEGVAL_CELL
  case (type = "L") .and. (.not. value)
    clr := NEGVAL_CELL
  otherwise
    clr := REGULAR_CELL
  endcase
return clr


/*——————————————————————————————————*/
```

```
function Navigate(b, k)
/*
   Establish array of navigation keystrokes and the cursor movement
   method to associate with each key. The array is comprised of
   two-element arrays containing the inkey() value of the key and a
   codeblock to execute when the key is pressed.

   This function gets passed a browse object and a potential
   navigation key. If the key is found in the array it's
   associated navigation message is sent to the browse.
   Function returns .t. if navigation was handled, .f. if not.
*/
local n

//  Made static so it doesn't get re-initialized on every call.
//  Due to Clipper bug of some sort it's not possible to directly
//  assign this array on the static statement line.  Perhaps this
//  will be fixed by the time you read this, if so you can eliminate
//  the if..endif and assign the array directly on the static
//  statement line.
//
static keys_
  if keys_ = nil
    keys_ := { ;
      {K_UP,         {|| b:up()       } }, ;  //  Up one row
      {K_DOWN,       {|| b:down()     } }, ;  //  Down one row
      {K_LEFT,       {|| b:left()     } }, ;  //  Left one column
      {K_RIGHT,      {|| b:right()    } }, ;  //  Right one column
      {K_PGUP,       {|| b:pageUp()   } }, ;  //  Up on page
      {K_PGDN,       {|| b:pageDown() } }, ;  //  Down one page
      {K_CTRL_PGUP,  {|| b:goTop()    } }, ;  //  Up to the first record
      {K_CTRL_PGDN,  {|| b:goBottom() } }, ;  //  Down to the last record
      {K_HOME,       {|| b:home()     } }, ;  //  First visible column
      {K_END,        {|| b:end()      } }, ;  //  Last visible column
      {K_CTRL_HOME,  {|| b:panHome()  } }, ;  //  First column
      {K_CTRL_END,   {|| b:panEnd()   } }, ;  //  Last column
      {K_TAB,        {|| b:panRight() } }, ;  //  Pan to the right
      {K_SH_TAB,     {|| b:panLeft()  } }  ;  //  Pan to the left
    }
   endif

   //  Search for the inkey() value in the cursor movement array.
   //  If one is found, evaluate the code block associated with it.
   //  Remember these are paired in arrays: {key, block}.
   //
   n := ascan(keys_, { | pair | k == pair[1] })
   if n <> 0
     eval(keys_[n, 2])
   endif

return (n <> 0)

/*-------------------------------------*/

function EditCell(b, fieldName, editColor)
/*
   General-purpose browse cell editing function, can handle all database
   field types including memo fields. If you want the edits to "stick"
   you must assign fieldblock()-style column:block instance variables.
   All editing, including memo-edit, is done within the boundaries of
   the browse window. On exit any appropriate browse cursor navagation
   messages are passed along.
*/
local c, k, clr, crs, rex, block, cell


  //  Retrieve the column object for the current cell.
  c := b:getcolumn(b:colPos)


  //  Create a field block used to check for a memo field
  //  and later used to store the edited memo back. It's
  //  done this way so you can have the browse window display
  //  a notation like "memo" rather than displaying a small
  //  hunk of the real memo field.
  //
  block := fieldblock(fieldName)
```

```
// Can't just "get" a memo, need a memo-edit.
if valtype(eval(block)) = "M"

  // Tell the user what's going on.
  //
  @ b:nTop, b:nLeft clear to b:nBottom, b:nRight

  @ b:nTop, b:nLeft say ;
    padc("Memo Edit: Record " +lstr(recno()) ;
        +', "'+ c:heading +'" Field', b:nRight -b:nLeft)

 @ row() +1, b:nLeft say replicate("-", b:nRight -b:nLeft +1)


  // Turn cursor on and perform the memo edit
  // using the specified color.
  crs := setcursor(1)
  clr := setcolor(editColor)
  cell := memoedit(eval(block), b:nTop +2, b:nLeft, b:nBottom, b:nRight)
  setcursor(crs)
  setcolor(clr)


  // If they didn't abandon the edit, save changes.
  // When passed a parameter, fieldblock-style code
  // blocks store the value back to the database.
  // Handiest darn thing they ever stuck in this language.
  if lastkey() <> K_ESC
    eval(block, cell)
  endif


  // We mussed up the entire window, tell TBrowse to clean it up.
  b:invalidate()

  // Re-read from database, since we edited it.
  b:refreshCurrent()


// Regular data type, do a GET/READ.
```

```
else

  //  Pass along any additional keystrokes.
  if lastkey() = K_CTRL_ENTER
    keyboard(chr(K_CTRL_Y))
  elseif (lastkey() > K_SPACE) .and. (lastkey() < 256)
    keyboard(chr(lastkey()))
  endif


  //  Create a get object for the field.
  cell := getnew(row(), col(), c:block, fieldName,, "W/N,"+editColor)


  //  Allow up/down to exit the read, and turn the cursor off.
  rex := readexit(.t.)
  crs := setcursor(1)

  //  Perform the read.
  readmodal({cell})

  //  Restore original cursor and read-exit states.
  setcursor(crs)
  readexit(rex)


  //  If user hit a navigation key to exit, do it.
  if Navigate(b, lastkey())

  //  If they pressed Enter, advance to next column.
  elseif lastkey() = K_ENTER
    if b:colPos < b:colCount
      b:right()
    else
      b:down()
      b:colPos := b:freeze +1
    endif
  endif
```

```
    //  We changed the field value and TBrowse doesn't know it.
    //  So we must force a re-read for the current row.
    b:refreshCurrent()
  endif

return nil

/*————————————————————————————————————————*/
// eof MaxiBrow.Prg
```

```
/*
  ERRSAVE.PRG:  Error object inspector and recorder.

  Author: Craig Yellick                    Thanks to Ted Means for
     Ver: 1.2a, 20-Apr-91                  supplying the little
                                           CurDrive() ditty in ASM.

  ______________________________

  To install this utility in your application the following lines must
  be executed prior to any errors that you want to trap.  This means
  they must be the first executable statements in your application if
  you want ErrorSaver() to be in effect for the entire application. The
  text filename is optional, if specified the file will be appended
  with error information.

        local defErr := errorblock()
        errorblock( { |e| ErrorSaver(e, defErr, ;
                   "<App Title>", "ERR.TXT") } )

  Compile with: /n /w /a

  ______________________________

 Note: ErrorSaver() calls CurDrive(), a small ASM routine that returns
  the letter of the current drive volume. An object file, CURDRIVE.OBJ,
  has been included. If you currently use a 3rd-party library which
  already supplies this function you can change the call to CurDrive()
  to use the library's syntax. If you prefer not to use the function
  you can delete the reference, it's not critical.

  ______________________________

  File Contents
  =============

  ErrorSaver( objError, [bError], [cAppTitle], [cFilename] )  ->  .f.

    Target function for error blocks.  Displays screen containing
    complete error information and scrolling program trace back window.
```

```
      If optional filename is specified the error information is appended
      to file.


    static Message( nTop, nLeft, [nDepth], [nWidth], :
                    [aStaticText], [cStaticColor],   :
                    [aVariText], [cColorVari] )  ->  nSelection

      General-purpose message display utility.  Many possible parameter
      combinations. See source code comments for details.


    static SetScrEnv( [aNewEnv] )  ->  aOldEnv

      Saves/restores screen environment to/from an array: cursor, color,
      row, column, screen contents.


    static XtoS( xValue )  ->  cValue

      Takes parameter of any data type and returns character string
      representation.


    static DosErrText( nError )  ->  cDescription

      Returns description of DOS error code number.

    ________________________________

  */


  /*
    []  A test of the error handler.

    Compile ERRSAVE.PRG to ERRSAVE.OBJ and link with CLIPPER.LIB,
     EXTEND.LIB and the supplied CURDRIVE.OBJ.

     When you run ERRSAVE.EXE it will immediatly bomb with a "missing
     database" error and display the ErrorSaver screen.  A file called
    ERR.TXT will also have a copy of the error information appended to
    the end of it.

    Comment-out the "#define TESTING" line and re-compile to prevent
   function Main() and the four "nesting" functions from being
   included in the ERRSAVE.OBJ you use in your applications.
*/

#define TESTING
#ifdef TESTING

        function Main()
          local defErr := errorblock()
          errorblock( { |e| ErrorSaver(e, defErr, ;
                        "Test Application", "ERR.TXT") } )
          ? "Here we go..."
          nest1()
        return nil

        function nest1()
          nest2()
        return nil
        function nest2()
          nest3()
        return nil
        function nest3()
          nest4()
        return nil
        function nest4()
          nest5()
        return nil
        function nest5()
          use WHERIZIT
        return nil
#endif

/* ---------------------------------------- */

/*
   Handy preprocessor directives.
```

```
*/


//  Convert integers to left-trimmed strings.
#define lstr(n)  (ltrim(str(n)))

//  Convert logicals to text.
#define YN(L)    if(L, "Yes", "No ")

//  Short-hand.
#translate ifempty(<a>, <b>) => if(empty(<a>), <b>, <a>)

/* ——————————————————————————————— */


function ErrorSaver(e, defError, appTitle, filename)
/*
 Display a screen containing everything known about the run-time error
  represented by the error object passed as a parameter.  If a filename
  is specified, append error information to the file.  If an application
  title is specified it will be displayed and written to the file.

  e          The error object containing run-time information.

  defError   The default error handler, required if you want
             to be able to pass the error along to Clipper's
             default error handler after inspecting it here.

  appTitle   Optional but important, this string will get displayed
             on the screen and written to the file. Put a version
             number here as well, it'll help with phone support.

  filename   Optional name of the file in which to record the error.
             If it already exists it will be appended.

*/

local errEnv, appEnv := SetScrEnv()
local varList_, trace_
local i, r, c, sel, argStr, argCnt, osDescr
```

```
/*
   Build array with procedure/line traceback. Start from 1, since we
   don't care about being down here in ErrorSaver() or where the
   errorblock was installed.
*/
i := 1
trace_ := {}
do while .not. empty(procname(i))
  aadd(trace_, procname(i) +" (" +lstr(procline(i)) +")")
    i++
enddo

//  Build list of arguments (if any)
if valtype(e:args) = "A"
  argStr := ""
  aeval(e:args, { |s| argStr += (XtoS(s) +", ")} )
  argStr := left(argStr, min(len(argStr) -2, 35))
  argCnt := lstr(len(e:args))
else
  argStr := "<none>"
  argCnt := "0"
endif

//  Build description of operating systen error
if e:osCode > 0
  osDescr := lstr(e:osCode) +": " +left(DosErrText(e:osCode), 35)
else
  osDescr := "0: n/a"
endif

varList_ := {"arg count          " +argCnt,               ;
             "args               " +argStr,               ;
             "canDefault         " +YN(e:canDefault),     ;
             "canRetry           " +YN(e:canRetry),       ;
             "canSubstitute      " +YN(e:canSubstitute),  ;
             "description        " +e:description,        ;
             "filename           " +e:filename,           ;
             "genCode            " +lstr(e:genCode),      ;
             "operation          " +e:operation,          ;
```

```
                "osCode              " +osDescr,              ;
                "severity            " +lstr(e:severity),     ;
                "subCode             " +lstr(e:subCode),      ;
                "subSystem           " +e:subSystem,          ;
                "tries               " +lstr(e:tries),        ;
                "_________ ",                                ;
                "Free memory   (0) " +lstr(memory(0)),        ;
                "Largest block (1) " +lstr(memory(1)),        ;
                "Run area      (2) " +lstr(memory(2))  }


  //  Display screen heading
  if valtype(appTitle) <> "C"
    appTitle := ""
  endif
  Message(1, 20,,, {padc(appTitle, 40), ;
                    padc("Run-Time Error", 40)}, "R+/B")


  //  If filename was specified, open it up and append error info.
  if valtype(filename) = "C"
    set alternate to (filename) additive
    set alternate on
    set console off
    ? replicate("=", 70)
    ?  "ErrorSaver: This run-time error logged on "
    ?? dtoc(date()) +" at " +time()
    if .not. empty(appTitle)
      ?  "Application: " +appTitle
    endif
    ?  "Operating system = " +os() +", network = "
    ?? ifempty(netname(), "<none>")
    ?  "Available diskspace = "
    ??  ltrim(transform(diskspace(), "999,999,999,999")) +" in "
    ??  curdrive() +":\" +curdir()
    ?  "PATH    = " +ifempty(gete("PATH"),    "<none>")
    ?  "COMSPEC = " +ifempty(gete("COMSPEC"), "<none>")
    ?  "CLIPPER = " +ifempty(gete("CLIPPER"), "<none>")
    //
    //  Add other DOS environment variables you might need to know about.
    //
    ? replicate("-", 70)
    ? "Traceback: Proc (Line)   Error Information"
    ? "~~~~~~~~~~~~~~~~~~~~~~   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
    for i := 1 to max(len(trace_), len(varList_))
      if i <= len(trace_)
        ? "  " +padr(trace_[i], 20)
      else
        ? space(22)
      endif
      ?? space(3)
      if i <= len(varList_)
         ?? varList_[i]
      endif
    next i
    set console on
    set alternate off
    set alternate to
  endif


  //  Display error object instance variables
  Message(4, 2, 19, 44, varlist_, "W+/B")

  //  Instructions for viewing traceback window.
  //  (If window isn't filled, no need to display this.)
  if len(trace_) > 7
    Message(15, 47, 8, 30, {"Use page-up and", ;
                            "page-down keys to", ;
                            "scroll through the", ;
                            "traceback window.", "", ;
                            "Press ESC to continue."}, "GR+/B")
  endif

  //  If window isn't filled, no need to wait for keystroke.
  if len(trace_) <= 7
    keyboard(chr(27))
  endif

  //  Display traceback in scrolling window
```

```
  Message(4, 47, 10, 30, ;                                   // Co-ordinates
         {"Traceback: Procedure (Line)"}, "W+/B", ;  // Static portion
         trace_, "W/B,B+/W")                         // Scrollable portion


  do while .t.
    sel := Message(15, 47, 8, 30, ;
                   {"Select..."}, "GR+/B", ;
                   {"Quit to DOS", ;
                    "Pass Error to Clipper", ;
                    "View Application Screen"}, "W/B,B+/W")
   do case
   case sel = 1
     exit
   case sel = 2
     if defError <> nil
       SetScrEnv(appEnv)
       return eval(defError, e)
     endif
   case sel = 3
     errEnv := SetScrEnv()
     SetScrEnv(appEnv)
     r := row()
     c := col()
     setcolor("W+*/R")
     @ 0,0 say "ERROR!"
     devpos(r,c)
     inkey(0)
     SetScrEnv(errEnv)
   endcase
  enddo

  //  Before returining, restore screen environment.
  SetScrEnv(appEnv)

return .f.

/* ---------------------------------- */
```

```
static function Message(r1, c1, rLen, cLen, ;
                        stat_, clrStat, ;
                        vari_, clrVari)
/*
   Display static message in box with scrolling variable section.
   Returns variable selection number, or zero if no selection made.

   Important note- the window size parameters are not the usual "four
   corners", you must specify a starting row/col and optionally a
   depth and width.  For example, (2,5,10,30) is a box 10 rows long
   and 30 columns wide starting at row 2, column 5.

    ╔═════════════════╗
    ║ Static line(s)  ║
    ║ :               ║
    ╟─────────────────╢
    ║ Scrolling line(s)║
    ║ : (optional)    ║
    ║ :               ║
    ╚═════════════════╝

    r1, c1   Required starting row and column.

    rLen     Optional number of rows down from r1, if not specified
             will be calculated based on maxRow() and larger
             of element count in stat_ and vari_ arrays.

    cLen     Optional number of columns over from c1, if not specified
             will be calculated based on maxCol() and longest
             element within stat_ and vari_ arrays.

    stat_    Array of strings which form static part of message.

    clrStat  Option color for static part of message.

    vari_    Optional array of strings which form the scrolling part
             of the message.

    clrVari  Optional color for scrolling part of message.
```

```
*/

local i, sel := 0, maxC := 0, maxR := 0, clr := setcolor()

  //  Determine length of longest string in stat_ and vari_.
  if cLen = nil
    aeval(stat_, { |s| maxC := max(len(s), maxC) } )
    if vari_ <> nil
      aeval(vari_, { |s| maxC := max(len(s), maxC) } )
    endif
    //  Add extra columns for box lines and spacing.
    maxC := min(maxCol() -c1, maxC +3)
  else
    maxC := min(maxCol() -c1, cLen)
  endif

  //  Determine number of rows required.
  if rLen = nil
    //  Add extra rows for spacing.
    i := if(vari_ = nil, 1, len(vari_) +2)
    maxR := min(maxRow() -r1, len(stat_) +i)
  else
    maxR := min(maxRow() -r1, rLen)
  endif

  //  Clear the area and draw box lines
  setcolor(clrStat)
  @ r1, c1 clear to r1 +maxR, c1 +maxC
  @ r1, c1 to r1 +maxR, c1 +maxC double
  if vari_ <> nil
    @ r1 +len(stat_) +1, c1 say "╟" +replicate("─", maxC -1) +"╢"
  endif

  //  Display static line(s)
  for i := 1 to len(stat_)
    @ r1 +i, c1 +2 say stat_[i]
  next i

  //  If variable portion specified, display in scrolling window.
  if vari_ <> nil
```

```
    setcolor(clrVari)
    sel := achoice(r1 +len(stat_) +2, c1 +2, ;
                   r1 +maxR -1, c1 +maxC -2, vari_)
  endif

  //  Restore original colors.
  setcolor(clr)

return sel


/* ----------------------------------------- */


static function SetScrEnv(restore_)
/*
   Return array containing current cursor, color, row, column
   and screen contents.  If an array is passed, restore the
   same elements based on the array's contents.
*/

// Save current screen environment
local old := { setcursor(), ;  //  1
               setcolor(),  ;  //  2
               row(), ;        //  3
               col(), ;        //  4
               savescreen(0,0,maxrow(),maxcol()) }  //  5

  if valtype(restore_) = "A"
    setcursor(restore_[1])
    setcolor(restore_[2])
    devpos(restore_[3], restore_[4])
    restscreen(0,0,maxrow(),maxcol(), restore_[5])
  endif

return old


/* ----------------------------------------- */
```

```
static function XtoS(x)
/*
   Takes parameter of any type and returns a string version.
*/
local s
  if     valtype(x) = "C"
    s := x
  elseif valtype(x) = "N"
    s := lstr(x)
  elseif valtype(x) = "D"
    s := dtoc(x)
  elseif valtype(x) = "L"
    s := if(x, ".t.", ".f.")
  endif
return s


/* ───────────────────────────────── */


static function DosErrText(n)
/*
   Return description of DOS error code.
   (Descriptions based on table D-1 in
   Clipper 5 Programming & Utilities Guide.)
*/
local descr_ := {"Invalid function number", ;  // 1
                 "File not found", ;  // 2
                 "Path not found", ;  // 3
                 "Too many files open (no handles left)", ;  // 4
                 "Access denied", ;  // 5
                 "Invalid handle", ;  // 6
                 "Memory control blocks destroyed (oh, my)", ; // 7
                 "Insufficient memory", ;  // 8
                 "Invalid memory block address", ;  // 9
                 "Invalid environment", ;  // 10
                 "Invalid format", ;  // 11
                 "Invalid access code", ;  // 12
                 "Invalid data", ;  // 13
```

```
, ;  // 14
"Invalid drive was specified", ;  // 15
"Attempt to remove the current directory", ;  // 16
"Not same device", ;  // 17
"No more files", ;  // 18
"Attempt to write on write-protected diskette", ;  // 19
"Unknown unit", ;  // 20
"Drive not ready", ;  // 21
"Unknown command", ;  // 22
"Data error (CRC)", ;  // 23
"Bad request structure length", ;  // 24
"Seek error", ;  // 25
"Unknown media type", ;  // 26
"Sector not found", ;  // 27
"Printer out of paper", ;  // 28
"Write fault", ;  // 29
"Read fault", ;  // 30
"General failure", ;  // 31
"Sharing violation", ;  // 32
"Lock violation", ;  // 33
"Invalid disk change", ;  // 34
"FCB unavailable", ;  // 35
"Sharing buffer overflow", ;  // 36
,,,,,,,,,,,,, ;  // 37-49
"Network request not supported", ;  // 50
"Remote computer not listening", ;  // 51
"Duplicate name on network", ;  // 52
"Network name not found", ;  // 53
"Network busy", ;  // 54
"Network device no longer exists", ;  // 55
"Network BIOS command limit exceeded", ;  // 56
"Network adapter hardware error", ;  // 57
"Incorrect response from network", ;  // 58
"Unexpected network error", ;  // 59
"Incompatible remote adapter", ;  // 60
"Print queue full", ;  // 61
"Not enough space for print file", ;  // 62
"Print file deleted (not enough space)", ;  // 63
"Network name deleted", ;  // 64
"Access denied", ;  // 65
```

```
                    "Network device type incorrect", ;  // 66
                    "Network name not found", ;  // 67
                    "Network name limit exceeded", ;  // 68
                    "Network BIOS session limit exceeded", ;  // 69
                    "Temporarily paused", ;  // 70
                    "Network request not accepted", ;  // 71
                    "Print or disk redirection paused", ;  // 72
                    ,,,,,,, ;  // 73-79
                    "File already exists", ;  // 80
                    , ;  // 81
                    "Cannot make directory entry", ;  // 82
                    "Fail on INT 24h", ;  // 83
                    "Too many redirections", ;  // 84
                    "Duplicate redirection", ;  // 85
                    "Invalid password", ;  // 86
                    "Invalid parameter", ;  // 87
                    "Network device fault", ;  // 88
                    ;
                    "Undefined or reserved error code!" } // +1

  /*
     Check that code number is within known upper limit,
     and that a description is available for it.
  */
  if (n > (len(descr_) -1)) .or. (descr_[n] = nil)
    n := len(descr_)
  endif

return descr_[n]

* eof ErrSave.Prg
```

Index

____ Please send me more information.

Name: ______________________________

Company: ______________________________

Address: ______________________________

City: ____________________ State: _____ Zip: __________

Country: __________ Telephone: __________ Fax: __________

Credit card #: ____________________ Expires: __________

Message Center Password: ____________ (3 - 10 characters please)

Disk Size Preference: ___ 5 1/4" ___ 3 1/2"

UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 9277 SALEM, OREGON

POSTAGE WILL BE PAID BY ADDRESSEE

GRUMPFISH, Inc.
Post Office Box 17761
Salem, Oregon 97305-9914

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT 871 REDWOOD CITY, CA

POSTAGE WILL BE PAID BY ADDRESSEE

M&T BOOKS
501 Galveston Drive
Redwood City, CA 94063-9929

135 92

Clipper 5: A Developer's Guide

Historically, Nantucket's Clipper® has been positioned primarily as a compiler for dBASE programs. However, the release of Clipper 5 has dramatically changed the rules of the game. With the addition of lexical scoping, an integrated preprocessor, four object classes, and other exciting features, Clipper has become a robust programming language with the ability to produce powerful applications.

This book provides Clipper developers of all levels with an excellent reference and tutorial on Clipper 5. It presents detailed discussions of Clipper's new features; tips for getting the most from Clipper's processor and debugger; tried-and-true methods for solving problems; numerous programming examples, and more.

Whether you are proficient in Summer'87 or are just starting out with Clipper, you'll reap immediate benefits from *Clipper 5: A Developer's Guide*.

Special features include:

- Extensive and thorough coverage of Clipper 5's new features, including lexical scoping; the preprocessor; code blocks; multi-dimensional and resizable arrays; and TBrowse, GET, Tbcolumn, and Error objects.
- Complete coverage of all Clipper 5.01 additions and changes.
- A thorough discussion of programming Clipper on a network.
- Numerous code examples for Clipper 5 developers of all levels.
- Practical tips and techniques for solving programming problems using Clipper 5.
- And last but not least, a style that is readable, understandable, and humorous.

All of the listings are available on disk in MS/PC-DOS format.

Joseph D. Booth is a seasoned Clipper user. He is the author of CLIPWKS, a soon to be released interface library between Clipper and Lotus/Quattro spreadsheets. **Greg Lief** is the author of the *Grumpfish Library* and creator/editor of *The Aquarium*, a disk-based Clipper technical journal. **Craig Yellick** is regularly published in various journals and newsletters, including *The Aquarium*, *Reference (Clipper)*, and *Compass for Clipper*. All three authors are regular speakers at the Clipper Developers' conference.

Why this book is for you– See page 1.

M&T BOOKS

M&T Books
501 Galveston Drive
Redwood City, CA 94063

53495
9 781558 512405
ISBN 1-55851-240-3

CHAPTER 25

The TBrowse Object Class

With the release of Clipper 5, Nantucket has adopted a new programming paradigm, the object class. Relative to other languages derived from dBASE, Clipper 5 has leapt ahead into the "modern" world of object-oriented programming (OOP). With respect to languages like C++, Clipper is only just starting down the road. Clipper 5 is not an OOP language in the strictest definition of the term, but it is a major step in that direction and an indication of what will be the future of the language. Clipper 5's implementation of objects presents new terminology and demands a new way of thinking.

Introduction to the Object Classes

The next chapters introduce the current Clipper object classes: Get, Error, TBrowse, and TBColumn. More classes are promised in the future. Object classes are a dramatic change from the traditional dBASE way of thinking, and unfortunately they've been implemented in Clipper 5 with woefully inadequate documentation and needlessly complex examples. Lest you get the wrong impression, we think Nantucket should be commended for taking our beloved language in this encouraging direction. Object-oriented programming is definitely the wave of the future and Nantucket is giving us an early opportunity to get comfortable with the jargon and concepts slowly rather than being completely overwhelmed with one giant leap. The following chapters explain the object classes using concepts with which you are already familiar, allowing you to make productive use of the object classes in your everyday programming.

We will learn about OOP backwards from the way most books and magazine articles approach it: We're going to completely ignore the OOP jargon and conceptual issues and plow right into the TBrowse class as if it were a traditional Clipper programmer's tool. The new jargon and concepts will be considerably easier to digest once you know how to use the tools because you'll be thinking about them in the context of living, breathing Clipper applications that you have created, not as abstract ideas.

Once you are comfortable with the concepts you can delve more deeply into these chapters and take advantage of features and functions far more powerful than could be accomplished with traditional programming techniques. Objects will without a doubt become one of your favorite features of Clipper 5.

Introducing Objects via the TBrowse Class

TBrowse is significantly easier to understand when stripped of all the fancy new terminology and exposed for what it really is: an outstanding browse utility! As you will see as this chapter progresses, TBrowse has more power and flexibility than the equivalents in dBASE-IV and FoxPro with a significantly better programming interface. The traditional dBASE-dialect way of doing things is to keep adding parameters and options to an already overburdened command. Take a look at the BROWSE syntax in dBASE IV or FoxPro. Yikes. Not that Clipper 5 is any less ambitious; the key difference is the way all those options and parameters are specified. Rather than passing 32 parameters and issuing a dozen application-wide SET commands to establish the browse, Clipper breaks the potentially complex situation into smaller and more manageable pieces. Consider what it takes in Summer '87 to get the much-maligned DBEDIT() up and running. We shudder to think what super_DBEDIT() would have looked like if Nantucket had copped out and just added more parameters.

We will start our introduction to the TBrowse object class by foregoing your indoctrination into the mysteries of object-oriented programming and instead just diving into the use of TBrowse as a tool for viewing database fields. Then, once you have a firm grasp of what TBrowse is all about, we will go back and see how OOP terminology and concepts help to describe and unify the new extensions of the Clipper language. We are of the opinion that most of us use Clipper to get production programming out the door, and the faster we get up to speed with its powerful new features, the better. OOP-speak, while important, should wait in line behind our employers or customers!

Although we face the risk of being branded as heretics, we are throwing back the curtain and exposing a "TBrowse Object" as a multi-dimensional array structure. When you create a TBrowse object you give it a name, and that name becomes the name of the array. When you "send messages" to the object you are assigning values to various array elements or calling a function that uses element values as data. The syntax for doing this has been adapted from OOP and is consequently a less painful introduction to such concepts. As you will see when you get more familiar with TBrowse, OOP concepts make this kind of programming much easier.

With that said, don't go overboard thinking about objects and arrays as being the same thing. They're not. In many respects they are similar, and for the purposes of introduction we're taking advantage of the similarities so we don't get bogged down in terminology.

On a fast track to our first TBrowse

Let's walk through the minimum steps for getting a TBrowse object up and running. The goal is to write the smallest possible program that browses any arbitrary database. This will form an easy-to-understand platform you can use for further experimentation. At the end of this section you'll have your first generic database browsing program.

Since a TBrowse object "lives" in what amounts to an array, the first step is to think up a name and assign it the basic TBrowse object structure.

```
browse := TBrowseDB()
```

We chose BROWSE as the name of the array. There's nothing special about the name; "RALPH" will work just as well.

We have skipped the four parameters that define the screen dimensions of the TBrowse; they default to the entire screen. TBrowseDB() is the same as TBrowseNew() except that it assigns database handling code blocks in the appropriate places in the object, which we will discuss in detail later. TBrowseDB() returns a mostly empty object. The "T" in TBrowse stands for table: TBrowse = Table-Browse. Some elements in the object deal with the entire browse, some deal with the columns that comprise a browse. At this point our browse object is a mostly empty shell with no columns and consequently nothing to display.

Each column in the browse screen is defined by a column object. Similar to a browse object, each column object is essentially an array that's eventually stored within the main browse object. The function used to return such an object is TBColumnNew().

```
column := TBColumnNew()
```

TBColumnNew() returns an object, so "column" is the name we have chosen for that object. At this point the column object is an empty shell. The minimum thing this object needs to be useful in a TBrowse object is a code block used to retrieve data for display. If you're confused by code blocks, don't despair. Most of the code blocks associated with TBrowse are easy to understand.

```
column:block := { || recno() }
```

The above line supplies a code block that returns the current record number. Our column object now "knows" how to retrieve data for the column, in this case the record number. The colon used to separate the object name from the word "block" is a completely new form of syntax in Clipper 5. In more traditional Clipper syntax this might have been written like so:

```
columnBlock(@column, { || recno() } )
```

The implication here is that the columnBlock() function is working with a parameter called column and a parameter that's a code block. Both of the previous column block lines are very different and distinct from the following, where the variable called column is being assigned a new value.

```
column := columnBlock( {|| recno()} )
```

Using the new syntax (which provides a level of power and efficiency that we're not even hinting at, yet), we are saying that of all the different values that a column object stores within itself, at this point we are providing just one of them — the code block used to retrieve data for the column.

```
column:block := { || recno() }
```

We still need to add the column we just defined to the currently column-less browse object.

```
browse:addColumn(column)
```

The addColumn() function knows precisely how to find an open spot in the main browse object and add the column object to it. Similar to the discussion about column:block, when you see browse:addColumn() you should think of addColumn() as one of several functions that know how to work with a browse object. In this case the addColumn() function adds a column object to the browse object and consequently needs a column object as a parameter.

Since we are going to browse records in a database it wasn't hard to assign a column for the record number, since all databases have record numbers. What do you do for fields? The next four lines assign all of the fields in a database as columns in the browse. Despite our original claim that this example would demonstrate only the "minimum necessary," this is a very important technique that you should understand early in the game. We could have written a few more TBColumnNew() functions, one for each field. But that would limit the resulting test routine to a single database. The following slightly more complex version will work with any database structure.

```
for n := 1 to fcount()
  column := TBColumnNew()
  column:block := fieldblock(field(n))
  browse:addColumn(column)
next n
```

We are looping through the fields, from the first to the last. The FIELDBLOCK() function returns a "get-set" code block for the field name returned by the FIELD() function. The term get-set denotes that the code block will retrieve, or get, the specified field value from the database, and can save, or set, a value back to the database. In this example we are interested only in the code block's ability to retrieve a value for display during the browse. We then use addColumn() to add the resulting column to the main browse object. If we were working with a database with name, address and city fields, the above loop could have been replaced by the following:

```
column := TBColumnNew()
column:block := fieldblock("Name")
browse:addColumn(column)

column := TBColumnNew()
column:block := fieldblock("Address")
browse:addColumn(column)

column := TBColumnNew()
column:block := fieldblock("City")
browse:addColumn(column)
```

A common question at this point is how does the browse object distinguish between column objects when in the example they all used "column" as a variable name? It's important to note that addColumn() does indeed add a column object to the browse object, but does not maintain any sort of relation between them. This will be more obvious in the next example.

If you want to eliminate the temporary variables you can use TBColumnNew()'s parameters which allow three lines of code to be combined into a single statement. TBColumnNew() takes two parameters: First, the heading for the column, and second, the code block used to retrieve the value. Since we're going for the absolute minimum in this initial example we'll skip the first parameter.

```
browse:addColumn(TBColumnNew(,  fieldblock("Name")))
browse:addColumn(TBColumnNew(,  fieldblock("Address")))
browse:addColumn(TBColumnNew(,  fieldblock("City")))
```

At this point we have the main browse structure defined with default database handling capabilities via a call to TBrowseDB() and several columns via calls to TBColumnNew() and addColumn(). So where is the browse? How do you get this wonder of modern programming technique on the screen? This is the really neat part of TBrowse, where the Nantucket developers demonstrate their considerable design talents. Using OOP-speak, you send the "stabilize()" message to your browse object. The stabilizing routine updates one portion of the display (usually one line of the table) and returns true if there is nothing more to update on the display. The easiest way to handle this is to wrap a "while loop" around it, as follows.

```
do while .not. browse:stabilize()
enddo
```

In rough terms, the loop will run once for each line visible in the table. The simple browse we established so far did not have any table dimensions so it fills the entire screen. On a 24-line screen browse:stabilize() would have to be called about 24 times. Why not just update the whole display? There are many reasons which we will not get into right now. Basically you can do whatever you want between each line displayed, for example, interrupting the entire process at line two rather than having to wait for all 24 lines. In this minimal example there is no way to take advantage of the feature, but it is easy to add. In the following example, any keystroke during the screen update will exit the loop.

```
do while .not. browse:stabilize()
  if nextkey() <> 0
    exit
  endif
enddo
```

The last minimum thing to do is allow the program operator to move around in the database. There are many different kinds of movement — up, down, left, or right; all the way to the top, bottom, left, or right; one screenful up or down; one column shifted to the left or right. TBrowse will do all the required scrolling in both directions. The following set of case statements will handle the minimum up/down and left/right movements. You need such a structure to associate each keystroke with a movement. (The items starting with K_ are manifest constants defined in the INKEY.CH include file).

```
#include "inkey.ch"
key := inkey(0)
do case
case key == K_UP       // up one row
  browse:up()
case key == K_DOWN     // down one row
  browse:down()
case key == K_LEFT     // left one column
  browse:left()
case key == K_RIGHT    // right one column
  browse:right()
endcase
```

All together now

We've covered all the steps required to use a TBrowse object to browse through any database.

1. Create a browse object and give it a name.
2. Create one or more column objects and add them to the browse object.
3. Display the browse window by calling the stabilizing function until there's nothing more to display.
4. Accept a movement keystroke and act on it.

Let's take a look at the final program, found in Listing 25.1. We have added the ability to pass a database filename in from the DOS command line, allowing you to more easily experiment with a variety of databases. If you have never programmed with TBrowse before you should take a break and experiment with this program. See Listing 25.1 for a sample screen generated by the MiniBrow program.

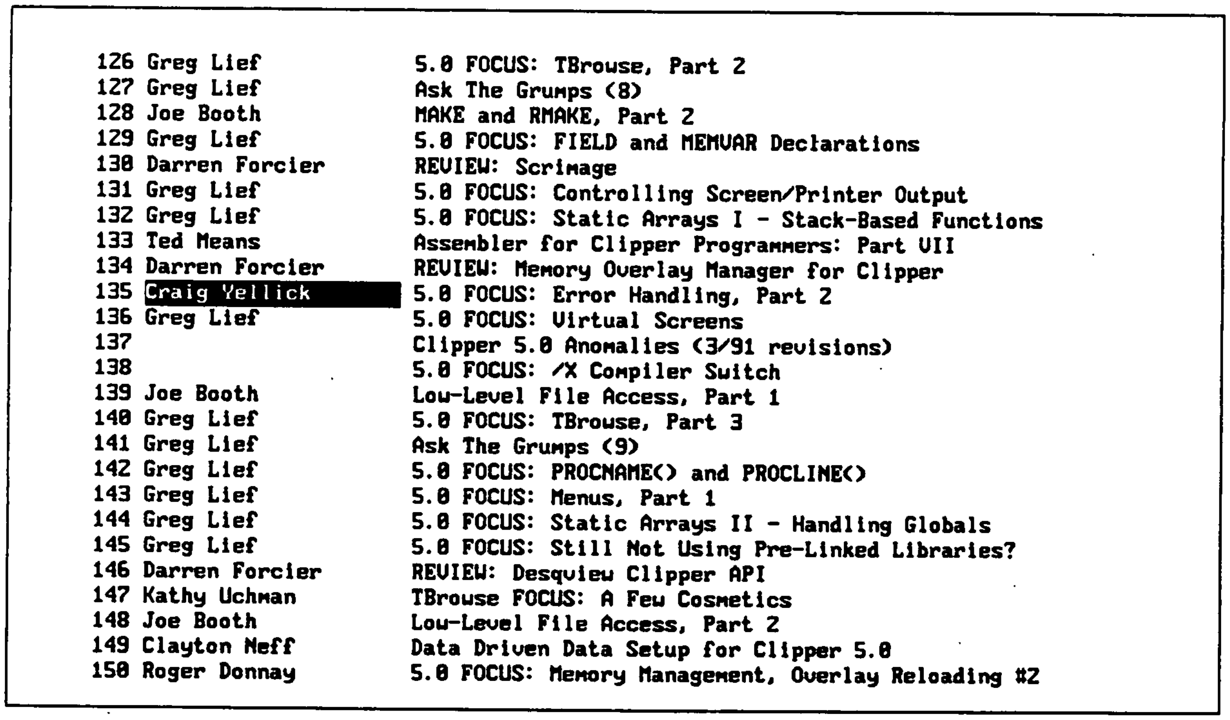
Listing 25.1 A complete TBrowse-based database browser

```
/*
   MiniBrow.Prg:  A database browser implemented with a minimum
   amount of code.
   Usage (from DOS):  minibrow datafile
   Compile with: /n /m
*/
#include "inkey.ch"

function Main(filename)
local column, browse, key, n

  //  Check for a DOS command line parameter
  if filename == nil
    ? "Must specify a database filename."
  //  Check that file exists.
  elseif .not. (file(filename) .or. file(filename +".DBF"))
    ? "File does not exist."
```

```
// Database checks out, clear screen and get down to it.
else
  @ 0,0 clear
  // Open the specified database.
  use (filename) new

  // Create a TBrowse object that knows how to deal with data
  // bases.
  browse := TBrowseDB()

  // Create a TBColumn object.
  column := TBColumnNew()
  // Assign a data retrieval code block to the column object.
  column:block := { || recno() }

  // Add the record number column to the browse.
  browse:addColumn(column)

  // For each field in the current database...
  for n := 1 to fcount()
    // Create a column object.
    column := TBColumnNew()
    // Assign the column a code block that retrieves
    // the field's value.
    column:block := fieldblock(field(n))
    // Add the column to the browse.
    browse:addColumn(column)
  next n

  // Keep looping as long as user wants to browse.
  do while .t.
    // Keep looping until all rows in window have been
    // displayed.
    do while .not. browse:stabilize()
      // Allow user to interrupt by pressing a key.
      if nextkey() <> 0
        exit
      endif
    enddo
```

```
      //  Wait for a keystroke.
      key := inkey(0)
      //  Move the pointer based on user's keystroke.
      do case
      case key == K_UP        //  Up one row
        browse:up()
      case key == K_DOWN      //  Down one row
        browse:down()
     case key K_LEFT          //  Left one column
       browse:left()
      case key == K_RIGHT     //  Right one column
        browse:right()
      case key == K_ESC       //  Done browsing
        exit
      endcase
    enddo  //  While browsing
    close database
  endif  //  File exists
return nil
* eof MiniBrow.Prg
```

Figure 25.1 Sample screen generated by MiniBrow.Prg.



```
126 Greg Lief          5.0 FOCUS: TBrowse, Part 2
127 Greg Lief          Ask The Grumps (8)
128 Joe Booth          MAKE and RMAKE, Part 2
129 Greg Lief          5.0 FOCUS: FIELD and MEMVAR Declarations
130 Darren Forcier     REVIEW: Scrimage
131 Greg Lief          5.0 FOCUS: Controlling Screen/Printer Output
132 Greg Lief          5.0 FOCUS: Static Arrays I - Stack-Based Functions
133 Ted Means          Assembler for Clipper Programmers: Part VII
134 Darren Forcier     REVIEW: Memory Overlay Manager for Clipper
135 Craig Yellick      5.0 FOCUS: Error Handling, Part 2
136 Greg Lief          5.0 FOCUS: Virtual Screens
137                    Clipper 5.0 Anomalies (3/91 revisions)
138                    5.0 FOCUS: /X Compiler Switch
139 Joe Booth          Low-Level File Access, Part 1
140 Greg Lief          5.0 FOCUS: TBrowse, Part 3
141 Greg Lief          Ask The Grumps (9)
142 Greg Lief          5.0 FOCUS: PROCNAME() and PROCLINE()
143 Greg Lief          5.0 FOCUS: Menus, Part 1
144 Greg Lief          5.0 FOCUS: Static Arrays II - Handling Globals
145 Greg Lief          5.0 FOCUS: Still Not Using Pre-Linked Libraries?
146 Darren Forcier     REVIEW: Desqview Clipper API
147 Kathy Uchman       TBrowse FOCUS: A Few Cosmetics
148 Joe Booth          Low-Level File Access, Part 2
149 Clayton Neff       Data Driven Data Setup for Clipper 5.0
150 Roger Donnay       5.0 FOCUS: Memory Management, Overlay Reloading #2
```

What about the OOP jargon?

Now that you understand TBrowse from a more traditional Clipper language standpoint, we will go back and associate the OOP terminology with the appropriate features. We feel this is a better way to introduce a new way of thinking and programming. You can get only so far ignoring the OOP concepts, so it is time to introduce them.

We've seen only one small example of the power of objects in Clipper 5, but it should be enough to show you why they're so important. There are two aspects to the major new step toward object-oriented programming in Clipper: ENCAPSULATION, which allows an object to wrap up some very useful and flexible functionality into one easily-handled package, and CLASSES, which allow you to create as many versions of any type of Clipper object as you may need.

- CLASSES: A class is not an object, it's the specifications for one type of object. TBrowse is one type object class just as date is one type of variable, and character is another. As we'll see next, no object can be created except as an instance of some pre-defined class. The main difference between Clipper 5 and true OOP languages resides in the current limitations of the Clipper classes (Get, Error, TBrowse and TBColumn). In a true-OOP Clipper we would have the full specification of the whole collection of variables and methods in each class, whether part of the interface or not. We could also define new classes and create modified versions of the existing ones. Such capabilities are slated as future enhancements to the Clipper language.

- INSTANCES: An instance is an object and an object is an instance. What the term "instance" emphasizes is that the object belongs to a class. "An instance of a class" is any object belonging to that class. A class may have zero, one or even one hundred instances. Each instance is a completely separate collection of variables (including exported instance variables, which we'll get to in a moment), all conforming to the same class specification.

- ENCAPSULATION: An object or instance is simply a collection of memory variables, with the one difference that the rest of the program (and the programmer) have absolutely no access to most, or even all, of this collection. "Encapsulation" means first that the collection of variables that we call an object works as a black box to the rest of the program. But then, how does the program use the object at all? This is the second aspect of encapsulation: An object has a well-defined interface that allows you, indirectly, to change or read its collection of variables, and, even more importantly, to command them to do something useful, the way the stabilize() function incrementally updates the window of a TBrowse object. Most of the interface to a normal object or instance is made up of functions, called methods. But some of the variables forming an object may also be directly accessible. In that case, they're called "exported instance variables". We'll cover both aspects of the object interface later on.

- MESSAGES: Whenever the rest of the program communicates with an object through its specified interface, we say the program "sends messages" to the object. With one exception that we'll see further on, all messages take the following form.

```
<black box> : <interface call>
```

That is, either

```
<instance name> : <method call>
```

or

```
<instance name> : <exported instance variables>
```

- METHODS: Methods are arguably the most important part of an object class's interface. An object class defines a set of functions that are associated with the class and provide services for the programmer. A method is always applied against a particular instance, often with additional parameters that affect the way the method operates. For example, the down() method is used to command the TBrowse object to move the pointer down one row in the window. The down() method is common to all TBrowse objects but operates only on the particular instance to which it is sending its message.

- CONSTRUCTORS: Defining a class does not create a single instance. It does not reserve the space in memory for even one collection of the kind it specifies. In order to do that, we need to use a special method of the class, called a constructor. That method creates the correct collection of variables and returns an address that is in turn assigned to an identifier. In that way, the indentifier is linked to one new instance of the class. Consider the following two lines:

```
browse1 := TBrowseNew()
browse2 := TBrowseDB()
```

These examples show that, contrary to other methods, constructors are not called with

```
<instance name> : <method call>
```

because the instance does not exist before the constructor is called. Also, the example shows that one class can have several different constructors. All constructors for a class create the same collection of variables. But usually constructors also initialize part of the collection. The difference between TBrowseNew() and TBrowseDB() is in the way they initialize an instance of the TBrowse class. We have also seen that constructors can take parameters, like TBColumnNew() accepting a data retrieval block, so that the programmer may override or add to the default initialization.

- EXPORTED INSTANCE VARIABLES: "Exported" means visible — a part of the instance's interface is visible to the programmer. Like the rest of the instance variables, exported ones are different for each instance. If you have two TBrowse instances, browse1 and browse2, browse1:freeze isn't the same variable as browse2:freeze, and may or may not have the same value. Some exported instance variables, such as freeze, can be both read and written by your program. Nantucket calls those you can read and write "assignable". The others cannot be modified by your program. For example, the browse object's hitBottom instance variable can only be read, you cannot assign it a value.

MiniBrow.Prg revisited

Let's take a quick run through the previous MINIBROW.PRG source code example with an eye toward the OOP concepts being used.

- CLASSES and ENCAPSULATION: Just by deciding to use TBrowse to program a database browser we've taken advantage of the encapsulated nature of the TBrowse class. TBrowse has all the tools we need to do some very ambitious programming, but most of the complexity is completely hidden from view.

- CONSTRUCTORS: The very first thing we did was use a pair of constructor functions, TBrowseDB() and TBColumnNew(), to create "instances" of the objects we need.

- INSTANCES: We created a single instance of a browse object and a variable number of column object instances depending on the number of fields in a database. Each instance has all the "rights and privileges" of a member of its class.

- MESSAGES: We sent a flurry of messages to the objects we created, describing exactly what each column should contain and how to react to various keystrokes.

- EXPORTED INSTANCE VARIABLES: These variables form the interface to an instance of an object. We used one of them to assign data retrieval code blocks to each of the columns.

- METHODS: We used the addColumn() method to add columns to the browse object, and several of the many navigation methods, like up() and down(), to provide services to the person using the browse.

Compiler and linker details

We should point out some technical issues regarding the way that objects, with their methods and instance variables, are implemented differently from the regular variable and function names to which they are frequently equated. Method names and instance variables are symbols just like function names and memory variables, with the important exception that they are created dynamically at run-time. The ramifications of this are important. Neither the compiler nor linker "sees" the method or instance variable names, so you will not get any compiler or linker warnings on obviously incorrect statements like

```
browse:iJustMadeThisUp := column:ObviousError(n)
```

or more subtle but equally incorrect statements like

```
browse:gotoTopBlock := { || RecPosition("top") }
```

where we accidentally misspelled browse:goTopBlock. You'll pay for your errors at run-time, at the moment Clipper encounters the incorrect method or instance variable name. Such is life with dynamic binding, a key component of object-oriented design. Only after your application calls TBrowseNew() or TBrowseDB(), the TBrowse class constructor functions, do the "correct" names become known. And even then, Clipper only knows about a problem when it is told to search a class's dictionary for a name and can't find it.

The implications for the programmer are staggering: You must vigorously test your code to ensure that every single line of object-oriented code is executed at least once at run-time. If in your testing you manage to avoid a section of code, you may never know it's a potential time-bomb waiting for the right logical conditions. This is not to imply that regular code should be any less subject to rigorous testing, only that object-oriented code merits special attention.

At run-time, Clipper will create an error with a generic error code of EG_NOVARMETHOD, or "no exported variable method", allowing you to detect the situation and possibly handle it rather than terminating the application. See Chapter 27, "The Error Class," for details on generic error codes and error handling.

What can objects do for me?

So far we have only scratched the surface of what TBrowse can do and we haven't even hinted at the other object classes. The remainder of this chapter is dedicated to a complete discussion of all the TBrowse features and functions. Then the following chapters discuss the Get and Error classes. You can use your understanding of the TBrowse object as a base to learn about the other object classes.

The TBrowse Object Class

The TBrowse class forms a general purpose table browsing utility that can be used with any form of data that is row and column oriented. Unlike the Summer '87 DBEDIT() function, TBrowse can be used to browse arrays and many other data structures, not just database files. TBrowse is perhaps the most accessible of the Clipper 5 object classes and an ideal way to introduce yourself to object-oriented programming (OOP).

This discussion explores the many features and functions of the TBrowse class and demonstrates ways you can use TBrowse objects in your everyday programming. To get the most out of TBrowse you must understand the basics of code blocks. Fortunately most of the code blocks associated with TBrowse are straightforward and don't require more than a passing familiarity with the syntax.

After reading this chapter you'll be able to create general-purpose database browsers as well as write functions for a variety of special browsing purposes. You'll also have all the tools you need to browse arrays and memo fields.

TBrowse overview

The Introduction to the Object Classes covers the basic concepts behind objects in general and TBrowse in particular.

Let's start by taking a look at the various things you can do with TBrowse and TBColumn objects as specified by the interface Nantucket provides. The process of getting a TBrowse up and running can be broken into four general steps:

1. Create a browse object and change browse-wide values as needed. Such values include, for example, the window coordinates and color scheme.

2. Create one or more column objects and change column-specific values as needed, then add the column objects to the browse object. For example, you can assign column headings and separators.

3. Display the browse on the screen.

4. Wait for a keystroke and then possibly take action on it. Loop back to step 3 to display the effect of the action. The action may affect things established in steps 1 and 2. Examples of actions are changing the color scheme, a column heading, or reacting to an attempt to scroll past the last record in the database.

All of the features of the TBrowse and TBColumn object classes can be divided into four broad categories defined by these steps. Many features are applicable to more than a single step. We've created these divisions purely as an aid in remembering how a feature is usually used. All instance variables and methods for both TBrowse and TBColumn will be covered in the course of these four categories.

We use the term "feature" to be deliberately vague. Your interface to Clipper's objects are the exported methods and instance variables described in the documentation. Some features are tied to a method, some to an instance variable, and others appear only when several different methods and/or instance variables are used in combination. If your view of the TBrowse object class is limited to an alphabetic list of methods and instance variables you will never see the incredible potential TBrowse offers, because a TBrowse object is a dynamic, run-time event and not just another complex function call like DBEDIT().

For the remainder of this chapter we will use the object name **b** when referring to methods and instance variables that are browse-related, and **c** for those that are column-related. To keep this overview section as concise as possible we will not supply comprehensive examples of how the methods and instance variables are used. After this overview we'll examine examples that cover all features in the context of a real browsing routine.

Step 1: Changing browse-wide values
The first step entails creating a new TBrowse object and possibly changing values that affect the entire browse. Most values have defaults that we find unacceptable, for example, the browse window occupying the entire screen. The following will be discussed:

Constructors
TBrowseNew()
TBrowseDB()

Instance Variables
b:autoLite
b:cargo
b:colorSpec
b:colSep
b:footSep
b:freeze
b:goBottomBlock
b:goTopBlock
b:headSep
b:nBottom

b:nTop
b:nLeft
b:nRight
b:skipBlock

There is a timing issue involved with respect to when an instance variable or method can legitimately be called. A good rule of thumb is that anything that deals with column-related information can't be referenced until at least one column has been added to the browse object. For example, don't assign anything to b:freeze until the column to be frozen has been added to the browse object. Since it usually doesn't matter what order things are done prior to the first b:stabilize() call, it's always safe to delay making assignments until after the general browse structure has been established. Create the browse object, fill it with columns, and *then* start making assignments to instance variables that affect the appearance or operation of the browse window.

TBrowseNew (nTop, nLeft, nBottom, nRight)
TBrowseDB(nTop, nLeft, nBottom, nRight)

TBrowseNew() is the main object constructor for the TBrowse class. It returns a new TBrowse object which is devoid of any knowledge about what you intend to browse and how you want it to look, essentially an empty shell. TBrowseDB() returns the exact same thing as TBrowseNew() except it defines the three database positioning code blocks (goTopBlock, goBottomBlock and skipBlock, discussed next). The following assign a new TBrowse object to brow1 and an object with basic database handling code blocks to brow2.

```
 //  Create two TBrowse objects.
brow1 := TBrowseNew()
brow2 := TBrowseDB()
```

Both constructor functions allow you to optionally specify the browse window coordinates. If you specify any of the window coordinate parameters they will be stored in the appropriate instance variables (discussed in a moment). The following creates brow3 as a new TBrowse object and assigns the indicated window coordinates.

```
 //  Create a TBrowse object and assign window coordinates.
brow3 := TBrowseNew(4, 5, 14, 75)
```

b:goTopBlock
b:go BottomBlock
b:skipBlock
These three instance variables store code blocks that are used to manipulate the source of the data being browsed. The variable names correspond to their Clipper database counterparts: GOTO TOP, GOTO BOTTOM, and SKIP. As previously described, the TBrowseDB() object constructor initializes all three variables with code blocks that will correctly handle a standard Clipper database. TBrowseNew() leaves the variables uninitialized.

b:goTopBlock and b:goBottomBlock are very simple code blocks. All they need to do is position the data source at the first or last position and return. Here are functionally equivalent versions of the TBrowseDB()-supplied code blocks.

```
b:goTopBlock    := { || dbGoTop() }
b:goBottomBlock := { || dbGoBottom() }
```

The code block simply calls the specified function and returns.

You may notice later that when we describe these code blocks we often use terms like data source positioning and avoid using database-specific terms like record and field. This is because one of TBrowse's most powerful features, and by extension part of the beauty and elegance of Clipper 5, is that TBrowse doesn't know or care about the source of the data. We can browse databases, arrays, text files, even data files created by other non-Clipper applications, simply by supplying code blocks that perform the required actions.

So much for the simple code blocks. b:skipBlock is a different matter entirely. This code block must be able to skip forwards or backwards through the data source, must be able to skip a specified number of positions, and must return the number it was actually able to skip. If the code block returns zero or a number less than was requested it assumes the data source hit the beginning or end of data during the skip operation.

Writing b:skipBlock functions will doubtless become an art form, as Clipper programmers strive for speed, flexibility and efficiency. The function in Listing 25.2 is a general-purpose database b:skipBlock function that you can use as basis for further experiments. It is the functional equivalent of the code block installed by TBrowseDB().

Listing 25.2 General-purpose database b:skipBlock function

```
function SkipRec(howMany)
/*
   General-purpose record skipping function for use with
   TBrowse skipBlocks. TBrowse will indicate how many records
   it wants to skip, and in what direction. Positive number
   indicates skip forward, negative indicates backwards. This
   function must return the number of records it was able to skip.
*/
local actual := 0

//  Negative = Move backward.
do case
case howMany < 0

//  Keep skipping backward until we skip the number
//  requested, or run out of records to skip.
do while (actual > howMany) .and. (.not. bof())
    skip -1
    //  Can't count the skip if we hit beginning-of-file.
    if .not. bof()
      actual--
    endif
enddo

//  Positive = Move forward.
case howMany > 0

//  Keep skipping forward until we skip the number
//  requested, or run out of records to skip.
do while (actual < howMany) .and. (.not. eof())
    skip +1
    //  Can't count the skip if we hit end-of-file.
    if .not. eof()
        actual++
    endif
enddo
```

```
if eof()
    skip -1
endif

//  No movement requested, re-read current record.
otherwise
    skip 0
endcase

return actual
```

The assignment for the code block that calls this function looks like the following.

```
browse := TBrowseNew()
browse:skipBlock := { |n| SkipRec(n) }
```

In later sections of this chapter we'll tackle the functions necessary for performing the same data source positioning operations for arrays and text files.

b:nBottom
b:nTop
b:nLeft
b:nRight

These four instance variables establish the coordinates of the window. They can optionally be assigned via parameters in the TBrowseNew() and TBrowseDB() constructor functions. They default to a value of NIL, which causes the window to occupy the entire screen, so they are not optional for most applications. The following lines assign a browse window that is 11 rows long and 70 columns wide.

```
//  Assign TBrowse window coordinates.
b:nTop    := 4
b:nLeft   := 5
b:nBottom := 14
b:nRight  := 75
```

It's important to point out that these define the overall window, not just the scrollable region. If you define column headings, footings and separators they will consume rows in the interior of the window. So, the 11 by 70 window could display as few as seven rows if all headings, footings and separators are used.

b:colSep
b:headSep
b:footSep

The column, heading, and footing separator variables contain the characters used to separate columns from each other and the column headings (if any) from the first row of data. The default column separator is a single space. There is no default heading or footing separator. The following illustrate some common values for these variables.

```
//  Narrow column spacing, single lines.
b:headSep :=   " ╤ "
b:colSep  :=   " │ "
b:footSep :=   " ╧ "

//  Wider columns, mixed single and double lines.
b:headSep :=   "═╦═"
b:colSep  :=   " ║ "
b:footSep :=   "═╩═"

//  Separators that can be output to any printer.
b:headSep :=   "---"
b:colSep  :=   " | "
b:footSep :=   "==="
```

It's convenient always to specify these values together, and to line up the values so you can visually inspect them for accuracy. These browse-wide values are used only when individual columns do not specify their own values.

b:colorSpec

Next to b:skipBlock, the b:colorSpec variable and associated color selection methods are often the most difficult TBrowse features to grasp, at least initially. TBrowse thinks of each cell (row/column intersection) as having one of two possible colors: highlighted and unhighlighted. The cell value gets displayed using the highlighted color when the browse cursor is in the cell, and the unhighlighted color when not.

TBrowse is compatible with the color setting scheme used by the rest of the Clipper commands and functions. This scheme is based on the SETCOLOR() function. (SETCOLOR() in turn is based on the SET COLOR TO command, which should have been ditched long ago along with other examples of crusty old dBASE-style syntax. We're stuck with it for the time being. But we digress.) TBrowse copies the current SETCOLOR() string into the b:colorSpec variable when a new object is created. Unless you change b:colorSpec your browse will use the same colors as appear in @..SAY..GET/READ situations.

```
set color to
? setcolor()              //  "W/N, N/W, N/N, N/N, N/W"

browse := TBrowseNew()
? browse:colorSpec        //  "W/N, N/W, N/N, N/N, N/W"
```

When presented with a b:colorSpec, TBrowse sees it as a list of possible cell colors. In the above example TBrowse thinks of the color string as having five possible cell colors. As we'll see later on when discussing c:defColor, c:colorBlock and b:colorRect(), TBrowse always deals with pairs of colors. By default TBrowse will display unhighlighted cells in the first color in the b:colorSpec list, and highlighted cells in the second color. We can assign b:colorSpec any number of colors; we are not limited to the five that the SETCOLOR() function uses. The following example illustrates a b:colorSpec with seven possible colors.

```
//  Assign TBrowse color scheme:
//
//                     1      2      3      4     5      6     7
brow:colorSpec := "BG/W, W+/B, GR+/R, R/W, W/RB, B/G, N/W"
```

We recommend always putting a comment line immediately above a b:colorSpec assignment when using literal color strings. The numbers will help you keep things straight. We'll come back to b:colorSpec in more detail when we cover the other color-related instance variables and methods.

b:autoLite

This variable controls whether or not the current cell is highlighted automatically. It defaults to .t., meaning TBrowse will call the b:hilite() and b:deHilite() methods automatically as you move the browse cursor. If you set b:autoLite to .f. you'll have to call the methods on your own. Most of the time you may as well leave b:autoLite set at .t. unless you have special highlighting needs. It's nice to have the control if you need it.

```
//  Turn automatic highlighting off.
brow:autoLite := .f.
```

b:freeze

The b:freeze instance variable is used to "freeze" one or more columns. Frozen columns will not be scrolled off the screen should you move the browse cursor into columns not currently visible. Without any columns frozen, the leftmost column would be scrolled out of the window to make room for the new column on the right of the window. b:freeze defaults to zero, meaning no columns will be frozen.

```
//  Freeze the first two columns
brow:freeze := 2
```

Column freezing is good user-interface design. The leftmost column should always be some kind of identifier that helps the user keep track of where they are in a wide set of columns.

Note: Do not assign a value to b:freeze until after columns have been defined and added to the main browse object. It gets ignored until there are columns to freeze.

b:cargo
This is a user-definable instance variable. b:cargo can be assigned any type of data and TBrowse will carry the value around for as long as the object exists.

b:cargo allows you to keep track of additional browse-related data and easily pass it along to functions that accept browse objects as parameters, as illustrated below.

```
brow:cargo := date()
? MyFunc(brow)          //  02/15/91

function MyFunc(obj)
return obj:cargo
```

b:cargo defaults to NIL and can contain any data type, including arrays with multiple dimensions. With an array you can maintain any arbitrary amount of data in a b:cargo variable.

Step 2: Changing column-specific values
After the main browse object has been established the next step is to create column objects and assign required values, and possibly change the optional values if we don't like the defaults. The finished column objects are then added to the browse object. We will discuss the following topics:

Constructor
TBColumnNew()

Method
b:addColumn()

Instance Variables
c:block
c:cargo
c:colorBlock
c:colSep
c:defColor
c:footing
c:footSep
c:heading
c:headSep
c:width

TBColumnNew(cHeading, bBlock)

TBColumnNew() is the only object constructor function for the TBColumn class. It returns a new TBColumn object with no useful default values unless the optional parameters are supplied.

```
//  Create a new TBColumn object.
col := TBColumnNew()
```

The TBColumnNew() function accepts two parameters: the column's heading and data retrieval code block. We'll discuss both instance variables next.

c:block

The data retrieval code block is the only required TBColumn object instance variable. The others can be left in their default states. The data retrieval code block is used primarily to read data from the data source, but it can refer to memory variables and expressions as well. When evaluated, the code block must return a value to display in the column. Each column's code block is evaluated repeatedly, once for each row in the browse window.

To create such a code block all you need to do is reference the database field name (or memory variable, as the case may be). A code block returns the value of the last expression evaluated, so in the case of a reference to a single field or variable name you get the current value.

```
//  Create a new TBColumn object.
col := TBColumnNew()

//  Assign a code block that retrieves
//  the current value of the LastName field.
col:block := { || LastName }
```

The value for the c:block instance variable can be passed as the second parameter to the TBColumnNew() constructor function. Since the retrieval block is never optional this is a handy place to assign it. The first parameter, as you might well guess from this example, is the column heading.

```
col2 := TBColumnNew("YTD Sales $", { || YTD_Sales } )
```

We recommend using TBColumnNew() to assign the heading and data retrieval code block at the same time. This helps keep the two items related and promotes the concept of self-documenting source code.

Since all the code block is concerned about is returning a value to display, you can do all sorts of interesting things. Here's an example of a calculation.

```
price := TBColumnNew("Item Price",  { || ItemPrice } )
qty   := TBColumnNew("Qty Ordered", { || OrderQty } )
total := TBColumnNew("Total Cost",  { || ItemPrice * OrderQty } )
```

You can also do a better job of interpreting the raw data values for your users. Here's an example that clarifies the meaning of a logical field.

```
taxStat := TBColumnNew("Tax Status", ;
            { || if(Taxable, "Add Tax", "Exempt ") } )
```

It's important to keep the two alternatives the same length, otherwise TBrowse will produce unpredictable results, unless you assign the column width directly via the c:width instance variable. Even then it's better design to return consistent lengths.

Here's an example that adds commas to a potentially large number. You can use the transform() function on any data type to produce a wide variety of special formatting.

```
c := TBColumnNew("1970 Population", ;
                 { || transform(Pop1970, "9,999,999,999") } )
```

You can even call a user-defined function to retrieve the data. Keep in mind that the function will be called repeatedly and should perform its duty as quickly as possible.

```
col := TBColumnNew("Calc'd Result", ;
                   { || CUST->(Results()) } )

function results()
local default := 2.319
select TABLE
seek CUST->Id
if TABLE->(found())
  return (CUST->Amount * ((default /TABLE->Factor) +1))
endif
return (CUST->Amount* default)
```

If you call a user-defined function from inside a data retrieval block, pay meticulous attention to work areas and the currently selected alias. If the function call leaves the wrong work area selected TBrowse will use the wrong database to do subsequent positioning operations.

This is as deep as we'll go regarding data retrieval code blocks for now. See the discussion on b:addColumn(), later in this section, for more details and a very important column definition technique using the FIELD(), FIELDBLOCK() and FIELDWBLOCK() functions.

c:heading
c:footing

A column's heading and footing are assigned via the c:heading and c:footing instance variables. As the names imply, the heading is displayed above the column and the footing, below. In the following example we use the footing to indicate the column number, 17, which will help the user keep track of where he is in a browse with many columns.

```
col:heading := "YTD Sales"
col:footing := "(17)"
```

The semicolon character (;) is used to indicate multiple lines within a heading or footing. The size of the heading or footing area in the window is determined by the column with the most lines.

```
//  Heading area in window will be three rows deep.
col1:heading := "One Line"
col2:heading := "Two;Lines"
col3:heading := "Three;Lines;Long"
```

As mentioned previously, the TBColumnNew() function accepts the column's heading as a parameter along with the column's data retrieval code block.

```
col := TBrowseNew("YTD Sales", { || YTD_Sales })
```

c:colSep
c:headSep
c:footSep

Any column may optionally replace the default values for the column and heading separators. Any column may also have an optional footing separator character. There are no default values for these instance variables. Here are some common values for these variables:

```
//  Narrow column spacing, single lines.
c:headSep := " ═╤═ "
c:colSep  := "  │  "
c:footSep := " ═╧═ "

//  Wider columns, mixed single and double lines.
c:headSep := "═╦═"
c:colSep  := " ║ "
c:footSep := "═╩═"

//  Separators that can be output to any printer..
c:headSep := "---"
c:colSep  := " | "
c:footSep := "==="
```

As we suggested with the browse-wide default values, it's convenient always to specify these values together and lined up for easy visual inspection.

c:defColor

This instance variable dictates which pair of colors from the b:colorSpec string will be used to display data in the column. The pair of colors is stored in c:defColor as an array with two elements. The default value is {1,2}, meaning the first color in b:colorSpec will be used for unhighlighted cells and the second color for the highlighted cell. Keep in mind that b:colorSpec is a character string, not an array, but is treated similarly to an array.

```
//  This column will use color #3 for unselected cells
//  and color #4 for the currently selected cell.
col:defColor := {3,4}
```

Earlier, when discussing b:colorSpec, we recommended always placing a set of comment lines immediately above the color specifications. Listed below is the previous example with some c:defColor-related additions.

```
/*
   Assign TBrowse color scheme:

   1:  Unselected, regular
   2:  Selected, regular
   3:  Unselected, numeric columns
   4:  Selected, numeric
   5:  Unselected, negative numbers
   6:  Selected, negative numbers
   7:  Used when highlighting a range of cells.
*/
//                    1     2     3      4     5     6     7
brow:colorSpec := "BG/W, W+/B, W/RB, RB/W, R/N, R/W, G/W"

#define CLR_REGULAR    {1,2}
#define CLR_NUMBERS    {3,4}
#define CLR_NEGATIVE   {5,6}
#define CLR_RANGE      {7,7}
```

Since almost all TBrowse color schemes are based on pairs of numbers, it's best always to assign them consistently. Odd numbers are unselected and even numbers selected: Each pair has a distinct meaning in the scheme. The above color scheme uses preprocessor #defines to make it easier to specify color pairs correctly and consistently, as illustrated below.

```
//  Create three columns.
col1 := TBColumnNew("Customer ID", { || Cust_ID } )
```

```
col2 := TBColumnNew("Name", { || CustName } )
col3 := TBColumnNew("Balance", { || CurBal } )

//  First two columns use "regular" colors,
//  third column contains numbers so use "numeric" colors.
col1:defColor := CLR_REGULAR
col2:defColor := CLR_REGULAR
col3:defColor := CLR_NUMBERS
```

The Customer ID column and the Balance column are both comprised of numbers, but the Balance column will be in a different color and thus more difficult to mistake for something else.

Using a block of comments above b:colorSpec and a set of #defines for the c:defColor pairs allows you to establish the entire color scheme in one place, which is a great way to keep it all straight.

c:colorBlock

c:colorBlock is a fascinating feature of the column object. Based on a code block that you supply, you can have a column display its values in different colors. In the color scheme listed previously we alluded to a "negative numbers" color pair. How do we tell when a number is negative and which color to use?

Each time the column's data retrieval code block fetches a field from the data source it passes the value to c:colorBlock, if one is defined. The c:colorBlock code block is expected to return a color pair array based on the value. If you don't specify c:colorBlock, then c:defColor is used (defColor stands for "default color pair"). Here's the previous example for the customer balance column, this time using c:colorBlock.

```
col3:colorBlock := { |n| if(n < 0, CLR_NEGATIVE, CLR_NUMBERS) }
```

For each row in the window, a value is passed to the code block. If the number is negative, the CLR_NEGATIVE color pair array (#defined previously as {5,6}) is used. If the number is positive, CLR_NUMBERS is used. Very simple but very powerful.

c:width

The c:width instance variable controls the width of the column when displayed on the screen. However, TBrowse uses the larger width of the heading or footing (if any) as the absolute minimum, independent of the c:width value. On the other hand, the highlighted browse cursor is based strictly on the c:width value and not on the actual width of the column as seen on the screen.

The c:width value defaults to the width from the first field read from the data source. If you use a fancy c:block that returns something other than the direct contents of a field, be careful always to return the same length.

```
//  Create column for the State field.
col := TBColumnNew("State", { || State } )

? col:width  //  2, because State field is two char wide.

//  Fancy block for logical values
col:block := { || if(IsMember, "Member", "Non-Member") }

? col:width  //  ?, depends on first value from data source!
```

In the above example the actual width of the column as it appears on screen will be at least five columns wide, due to the column heading. The column may actually be wider than five columns if, for example, the column separator characters include spaces on either side of a line. TBrowse centers the data within the column.

TBrowse left-justifies character data, right-justifies numbers, and centers logical values within each column.

c:cargo
This is another user-definable instance variable, similar to the browse object's b:cargo. The column object's c:cargo can be assigned any type of data and TBrowse will carry the value around for as long as the object exists. A common use for c:cargo is for help messages specific to each column.

```
col7 := TBColumnNew("Due", { || InvAmt } )
col7:cargo := "Total amount due on current invoice."

col8 := TBColumnNew("Balance", { || CurBal } )
col8:cargo := "Customer's current balance as of last statement."
```

c:cargo defaults to NIL and can contain any data type, including arrays with multiple dimensions. c:cargo can contain an arbitrary amount of data if you use an array.

b:addColumn(oColumnObject)
b:addColumn() is a browse method and not a column method, but we're covering it here because it operates on column objects. You call b:addColumn() once for each column you want in the browse. The columns will be displayed in the browse window in the order they are added to the browse object.

```
//  Create a browse with three columns:
//  Record #, First Name, Last Name.
//
brow := TBrowseDB(10, 25, 19, 55)
c1 := TBColumnNew("Record #", { || recno() } )
c2 := TBColumnNew("First Name", { || First } )
```

```
c3 := TBColumnNew("Last Name", { || Last } )
brow:addColumn(c1)
brow:addColumn(c2)
brow:addColumn(c3)
```

Creating columns more efficiently

It's tedious to have to type code blocks over and over each time you create a column that's based on a database field name, which is the case for most browsing situations. This is especially true when you need a dozen (or more!) columns. Even worse, there are situations when you don't know the field names in advance.

Fortunately there are several Clipper functions that make this process considerably easier: FIELD(), FIELDBLOCK() and FIELDWBLOCK().

FIELD() returns the name of the field specified by number. If there are ten fields, FIELD(7) is the name of the seventh field. You can use FCOUNT(), the database field count function, to tell you how many fields a database contains. Here's an example that lists all the field names in the SALES database.

```
use SALES new
for i := 1 to fcount()
  ? field(i)
next i
```

FIELDBLOCK() and FIELDWBLOCK() return a code block that can be used with c:block, based on the field name you send as a character string parameter. Don't pass the actual field, these functions will attempt to use the current value of the field rather than the field's name. The FIELDBLOCK() function returns a code block that will work in any work area, while FIELDWBLOCK() returns a code block that is tied to a specific work area. The following shows FIELDBLOCK() being used to return a code block for the ZipCode field.

```
column:block := fieldblock("ZipCode")
```

The problem here is that we are referencing a symbol (the ZipCode field name) that the compiler never sees. This is not good programming practice and is likely to cause confusion in complex applications with many databases open at the same time. Use the FIELD statement to unambiguously declare ZipCode to be a field name found in a specific database.

```
//  At top of routine, tell compiler that ZipCode
//  is a field name found in the CUST database.
field ZipCode in CUST

//  Later, in body of routine...
column:block := fieldblock("ZipCode")
```

FIELDWBLOCK() is used when you might have more than a single work area in use and want to guarantee that the column displays data from only a specific work area, not the work area that happens to be selected at the moment.

```
column:block := fieldWblock("City", 3)
```

Listed below are code blocks functionally equivalent to the ones that FIELDBLOCK() and FIELDWBLOCK() return.

```
//  Code block returned from fieldblock().
{ | x | if(x == nil, ZipCode, ZipCode := x) }

//  Code block returned from fieldWblock().
{ | x | if(x == nil, 3->City, 3->City := x) }
```

These are no doubt more complicated then you'd think they need to be, and as far as TBrowse is considered, they are. These code blocks not only return the current value of the specified field, but are also capable of replacing the current value with the parameter passed to the code block when it's evaluated. TBrowse does not pass any parameters to the data retrieval code blocks so it's safe to assume the value will always be NIL and no data will be changed. These code blocks are very useful for other purposes which we'll discuss in the second half of this chapter.

Recommendation: When using the FIELDBLOCK() function always be careful about conflicts with other field names. We've previously recommended that you never refer to a work area number directly, so using FIELDWBLOCK() is not a completely satisfactory solution unless you take pains to reference the work area number indirectly.

```
//  This DOES NOT work, can't use alias.
col:block := fieldblock("CUST->Name")

//  This is about as "safe" as you can make it.
field Name in VENDOR
col:block := fieldblock("Name")
```

Let's use the FIELD() and FIELDWBLOCK() functions to create columns for every field in any database. This example also illustrates how to use FIELDWBLOCK() without having to refer to a work area number directly.

```
brow := TBrowseNew(5,10,20,70)

use SOMEDATA new
for i := 1 to SOMEDATA->(fcount())
  col := TBColumnNew()
  col:heading := field(i)
  col:block := fieldWblock(field(i), select("SOMEDATA"))
  brow:addColumn(col)
next i
```

If you prefer to write code that uses the fewest possible temporary variables you can eliminate col completely, although that will decrease the readability and the likelihood of it being typed correctly.

```
use SOMEDATA new
```

```
for i := 1 to SOMEDATA->(fcount())
  brow:addColumn( ;
     TBColumnNew(field(i), ;
       fieldWblock(field(i), select("SOMEDATA")) ))
next i
```

Step 3: Displaying the Browse

Up to this point, despite all the object creating and variable assignments, nothing has been displayed on the screen. Once the browse object has been filled with the desired column objects, the next step is to display the actual browse window and fill it with data. The way this occurs is very simple, but it's an important concept and differentiates the TBrowse class from the more traditional techniques used by functions like DBEDIT(). The following will be discussed:

Methods

b:configure()

b:invalidate()

b:refreshAll()

b:refreshCurrent()

b:stabilize()

Instance Variable

b:stable

b:stabilize()

The b:stabilize() method is perhaps the most underappreciated method in the entire TBrowse class design. b:stabilize() is used to display and update the contents of the browse window. When you call b:pageDown(), for example, the entire window must be updated to reflect a new whole page of data. When you call b:up() or b:down(), in contrast, only a single row gets updated. Rather than making a call to a function that attempts to update the entire browse window at once, each call to b:stabilize() results in the update of one small portion of the window. This corresponds roughly to one row of data. To display a window with, for example, ten rows of data, b:stabilize() would need to be called at least ten times.

Sometimes only a single row in the window needs to be updated, sometimes it's the entire display. Rather than hardwiring a specific number of calls you instead take advantage of b:stabilize()'s return value. It will return .f. if TBrowse has more to do, or .t. if the window is finished being updated. The following will keep looping until the window is completely updated, or to use TBrowse terminology, is completely stable.

```
do while .not. browse:stabilize()
enddo
```

While the above loop does indeed handle the stabilizing process correctly it isn't taking advantage of the real reason the updating is done incrementally in the first place. Here's a better version:

```
do while .not. browse:stabilize()
  if nextkey() <> 0
    exit
  endif
enddo
```

This loop will exit before the window is fully stabilized if a keystroke is detected in the keyboard buffer. This allows the user to hit additional navigation or other keys without having to wait for the entire browse window to be updated. You can see this effect by hitting PageDown several times in rapid succession — rather than filling the entire screen with data, the browse will leap ahead another page and attempt to display it instead of the previous one. If you hit the key fast enough only a single row of data will be displayed before the loop is exited. The b:stabilize() method is not responsible for moving the record pointer (or whatever it's called in the data source you are browsing), only for displaying the data. When you call a navigation method, TBrowse handles the data source positioning independent of the display. The positioning methods communicate with the b:stabilize() method via the b:stable instance variable. This is not too shabby for what appears to be a simple little method.

There's a certain type of reader who wondered if the previous example could have been written with fewer lines. Here's what he was thinking.

```
do while (!browse:stabilize()) .and. (nextkey() # 0)
enddo
```

A common question at this point is, why use NEXTKEY() instead of INKEY()? The INKEY() function actually removes the keystroke from the keyboard buffer, while NEXTKEY() takes a peek but leaves the keystroke right where it is. We'll need the value of the keystroke in order to interpret it for navigation and whatnot, so if we used INKEY() we'd need to store the value to a variable. Some programmers prefer this method. Here's an example:

```
do while .not. browse:stabilize()
  if (key := inkey()) <> 0
    exit
  endif
enddo
```

This example takes advantage of Clipper's handy ability to store a value to a variable as part of an expression, saving us the trouble of assigning INKEY() to key on one line and then testing it on the next. Some find this hard to follow, and not without justification. Still others would go even further and write the same loop like so:

```
do while (!browse:stabilize()) .and. ((key := inkey()) # 0)
enddo
```

Earlier we mentioned that b:stabilize() is not directly linked to data source positioning or navigation operations (although it is affected by them), so it really has no need for the actual value of the keystroke. Just knowing one is pending is all you have to be concerned about during the stabilizing process. It's better that the source code related to keystroke handling be responsible for retrieving keystrokes from the keyboard buffer. Has this point been sufficiently run into the ground? Good. Let's continue.

b:stable

This instance variable simply reflects the status of the window. If b:stable returns .f. the window needs to be updated. If it returns .t. the window is stable. Any call to a navigation method, like b:Down(), causes b:stable to return .f. According to the Nantucket documentation, a browse window is stable when the following conditions are met: All data has been retrieved and displayed, the data source is positioned to match the placement of the browse cursor, and the current cell is highlighted.

```
do while .not. browse:stabilize()
enddo
? browse:stable    // .t.
browse:goBottom()
? browse:stable    // .f.
```

b:refreshAll()
b:refreshCurrent()

These two methods mark either the current row, or all rows, as being in need of updating. The b:stable variable will be set to .f. and the rows affected will be updated during the next stabilizing loop.

```
//  Mark current row as needing an update.
browse:refreshCurrent()

//  Mark all rows as needing an update.
browse:refreshAll()
```

Since TBrowse maintains such internal status on its own in order to automatically handle navigation calls like b:goTop() and so on, it may not be immediately obvious why you would want to mark a row (much less all the rows) for updating. As was touched upon briefly in the discussion of the c:block variable, TBrowse does not maintain a constant link between the data source and the values currently displayed on the screen. A non-TBrowse event may cause one or more rows in the data source to be changed. By calling the b:refreshAll() method you assure that the entire display will reflect the values in the data source. An example of such an event follows:

```
//  Display browse window with inventory prices.
do while .not. browse:stabilize()
enddo

//  Increase all prices by 25%.
replace all invent->price with invent->price * 1.25

? browse:stable  //  .t., as far as TBrowse knows.

browse:refreshAll()
? browse:stable  //  .f., due to refreshAll().

//  Update browse window with new inventory prices.
do while .not. browse:stabilize()
enddo
```

Unless a call is made to b:refreshAll(), the browse in the above example would still reflect the prices before the REPLACE command occurred. However, any navigation method that requires the entire window to be updated would end up using the correct prices, because data is read from the data source via the data retrieval code blocks during the stabilization process.

TBrowse works this way so you can take complete control over the timing of screen updates. It'd be pointless for TBrowse to slow down data manipulation operations just so the screen stays accurate at every instant. You can do all the manipulation you need and then request a browse window update when you want it. If you actually do want the browse window to be constantly up to date you can call the b:refreshCurrent() or b:refreshAll() methods and then stabilize the display. The point is, you are in control of the timing.

b:invalidate()

The b:invalidate() method is similar to b:refreshAll() in that b:invalidate() forces TBrowse to redisplay the entire window, including headings and footings, during the next stabilization process. The difference is that with b:invalidate() data is not read from the data source. This may result in significantly faster window updates if there's lots of data or a slow data source. b:invalidate() should be used in situations where you've changed something on the screen that isn't related to the values in the data source and need to have the window displayed again.

```
//  Display complex browse window with lots of data.
do while .not. browse:stabilize()
enddo

//  Call function that messes up the screen.
@ 0,0 clear
MessyFunc()

? browse:stable  //  .t., as far as TBrowse knows.

browse:invalidate()
? browse:stable  //  .f., due to invalidate().

//  Now window is quickly and completely redrawn.
do while .not. browse:stabilize()
enddo
```

In the above example the next time the stabilization process is performed, the browse window will be drawn completely, but faster than if TBrowse had to read all the data from the data source.

b:configure()

This browse method is similar to b:refreshAll() in that it causes TBrowse to react to changes in the browse environment. A call to b:configure() is very "expensive" with respect to processing time. It has TBrowse perform a complete reexamination of all of the instance variables for the overall browse and all of the individual columns, then make any internal adjustments needed to react to changes. The window will not be updated, however, until the next stabilizing process.

The configuration process is automatic in most circumstances. Calling b:configure() on your browse is normally required only when making direct changes to column object instance variables. TBrowse does not maintain permanent links between a browse object and the set of column objects of which it is comprised: If you alter a column's instance variables the browse object does not "see" the changes until you issue a b:configure().

This process of changing a browse or column instance variable, calling b:configure(), and then seeing the results during stabilization is broken into three distinct steps for a very good reason. It allows you to alter any number of values but have the actual changes performed all at once. If changes were reflected immediately, the display might jump around or do other disconcerting things.

Step 4: Taking action on keystrokes

The browse stabilizing process fills the window with as much data as can fit, then stops. However, a browse is a user-interface tool and is expected to do considerably more than just display one screen's worth of data. Like any other user-interface tool, a browse is manipulated by keystrokes, so the final step in the process is to accept keystrokes and take action on them. Actions take the form of navigation (moving the browse cursor around within the confines of the browse), configuration (altering column-specific or browse-wide values), and status (checking on things the browse keeps track of). We'll divide the remaining methods and instances into three categories: Navigation, configuration, and status.

Keep in mind that, as was mentioned in the previous steps, most of the browse-wide and column-specific values can be changed at any time and not just when the object is first created.

Navigation Methods

b:down()
b:end()
b:goBottom()
b:goTop()
b:home()
b:left()
b:pageDown()
b:pageUp()
b:panEnd()
b:panHome()
b:panLeft()
b:panRight()
b:right()
b:up()

Status Instance Variables

b:colCount
b:colPos
b:hitBottom
b:hitTop
b:leftVisible
b:rightVisible
b:rowCount
b:rowPos

Configuration Methods

b:colorRect()

b:delColumn()

b:deHilite()

b:getColumn()

b:hilite()

b:insColumn()

b:setColumn()

Navigation in general

TBrowse maintains an internal browse cursor. This cursor always remains within the boundaries of the browse window. A unique cursor is associated with the browse object, and is completely distinct from the common screen cursor which is controlled via the SETCURSOR() function and tracked via the ROW() and COL() functions. The navigation methods do not actually move the browse cursor on the screen; the effects of navigation calls are seen only after a stabilization process is complete. The navigation methods will cause the browse to need stabilization on their own; as a rule you do not have to worry about it unless you manipulate the screen directly or move the data source position on your own.

```
//  Jump to left-most column in browse.
b:panHome()

//  Move down one row.
b:down()

//  Browse cursor does not move on screen until
//  this stabilization loop is complete.
do while .not. b:stabilize()
enddo
```

Reexamination 90/005,727

The Original

Page 1094 of Part V

Is Missing

Initial : AR
Date : 6-8-2000

such a pain to accomplish with the much maligned and now obsolete DBEDIT() function can be implemented very easily with TBrowse. See the later section entitled "A TBrowse Toolbox" for complete details.

b:panEnd()
b:panHome()

b:panEnd and b:panHome jump the browse cursor to the very first or very last column. This is in contrast to the "plain" b:home and b:end methods which only jump within the currently visible set of columns. The browse window will be scrolled as needed to accommodate the jump. After a b:panHome, for example, you are certain to be positioned in column number 1. Be aware that this includes "frozen" columns, so after a b:panHome be sure to move the column position back to the right, past the frozen columns if you don't want the browse cursor to stop there.

```
//  Check to see if a recent navigation method
//  caused browse cursor to enter the frozen columns.
//  If so, move it back out.
if brow:colPos < brow:freeze
  brow:colPos := brow:freeze +1
endif

//  Stabilize the current window.
do while (.not. brow:stabilize()) .and. (nextkey() == 0)
enddo
```

b:panLeft()
b:panRight()

These two methods scroll the browse window left or right but leave the browse cursor in the same relative column position. Sort of handy if you have lots of columns and want to stay in one spot within the window while you scroll columns left or right. Note that it's not always possible to do this, so there's no guarantee that the cursor stays in exactly the same place.

Status in general

The browse object instance variables covered in this section are used to inquire about various parts of the browse object. Most of the time these variables should not be relied upon until immediately following a stabilization process. Exceptions are b:colCount and b:rowCount.

b:colPos
b:rowPos

b:colPos and b:rowPos return the current position of the browse cursor within the browse window, not the position on the screen. Column headings, footings, and separator rows are not considered in the row position. b:rowPos refers only to visible rows and has nothing to do with the number of records in the data source. b:colPos, however, keeps track of the actual column within the entire browse, not just the ones visible on the screen.

```
//  A status message that's helpful when experimenting.
@ 0,0 say "Browse cursor is at row " +ltrim(str(brow:rowPos))
?? ", column " +ltrim(str(brow:colPos))
```

It may surprise you that both b:colPos and b:rowPos are assignable. You can position the browse cursor directly where you want it. Most of the browse status variables are assignable.

```
//  Put browse cursor in last column in center row.
brow:colPos := brow:colCount
brow:rowPos := (brow:rowCount / 2)
```

It is good practice to call on b:invalidate() to force a screen update in the event that you placed the browse cursor in a column that isn't currently visible. If you rely on the side effects of some of the other regular browse navigation methods you may frequently find the browse window left in a messed-up state.

b:colCount
b:rowCount

These two instance variables, which return the number of rows or columns defined in the browse, are based on the way the browse was created and will remain accurate unless rows or columns are added. A call to b:configure() will guarantee that these values are correct. b:configure() was discussed in a previous section.

```
//  Helpful message to display while experimenting.
@ 1,0 say "Row count: " +ltrim(str(brow:rowCount))
?? ", Column count: " +ltrim(str(brow:colCount))
```

Unlike many of the other browse status variables, the row and column counts are not assignable.

b:hitBottom
b:hitTop

b:hitTop and b:hitBottom are logical flags updated by the stabilization process and indicate whether or not the most recent navigation method attempted to read past the beginning-of-file or end-of-file. For example, calling b:up() or b:pageUp() when you are already positioned at the first record in the data source will cause b:hitTop to be .t. after the next stabilization process.

```
//  Stabilize the browse window display.
do while (.not. brow:stabilize()) .and. (nextkey() == 0)
enddo

//  Another helpful status message to display while experimenting.
@ 2,0 say "Hit top: " +if(brow:hitTop, "Yes","No ")
?? ", Hit Bottom: " +if(brow:hitBottom, "Yes","No ")
```

b:leftVisible
b:rightVisible

These instance variables are used to determine what portion of the browse columns are currently visible. b:leftVisible returns the left-most visible column number, excluding any "frozen" columns. b:rightVisible returns the number of the right-most visible column. These two values are useful for displaying "more" indicators in conjunction with the browse window, for example, displaying an arrow on the right when the last column isn't currently visible, alerting the end-user that there's more information available. The following logic illustrates how to detect such situations:

```
//  This expression is true if the left-most column
//  is not currently visible.
//
if b:leftVisible > (b:frozen +1)
  //  More columns to the left
endif

//  This expression is true if the right-most column
//  is not currently visible.
//
if b:rightVisible < b:colCount
  //  More columns to the right
endif
```

If frozen columns are not being used, the b:leftVisible logic can be simplified to the following.

```
if b:leftVisible > 1
  //  More columns to the left
endif
```

Configuration in general

This final group is a catch-all for TBrowse methods that don't fit well anywhere else. This by no means diminishes their importance! In fact, b:getColumn() and b:setColumn() are indispensable methods for the construction of dynamic browse windows that can react to the user at run-time. They are grouped here because they usually can't be called with any useful effect until after the browse window is constructed and displayed.

It's also important to point out that all the browse or column methods and instance variables that have been discussed so far can also be changed at any time, not just the few methods listed here. TBrowse is a complete dynamic user-interface tool. Everything is available to you for changes, additions or deletions at any time.

b:getColumn(nColumn)
b:setColumn(nColumn, objColumn)

b:getColumn() and b:setColumn() are the workhorses of any dynamic browse where you do more than just allow the user to scroll around the window. b:getColumn() returns the complete column object for the specified browse column. b:setColumn() reverses the process and replaces a column with the specified column object. TBrowse will not react to changes made via b:setColumn() unless you use b:configure() to force it to examine all of its internals. Then you must perform a stabilization process to actually see any results on the screen.

The example in Listing 25.3 increases or decreases all column widths in a browse object, using both the b:getColumn() and b:setColumn() methods.

Listing 25.3 Function that makes all columns in a browse more wide or more narrow by a specified amount

```
function Widen(objBrow, nHowMuch)
/*
   Given a browse object, make all columns wider
   by specified amount. (Negative = more narrow).
   Prevents column from being smaller than 1 or
   larger than the width of the window. Note: Browse
   object must have window coordinates defined.
*/
local i, objCol, wid

//  Loop once for each column in the browse.
for i := 1 to objBrow:colCount
```

```
   //  Extract a column object.
   objCol := objBrow:getColumn(i)

   //  Add the amount to widen or narrow
   //  to the column's current width.
   wid := objCol:width + nHowMuch

   //  Trap for extremes, less than 1 or
   //  greater than width of browse window.
   objCol:width := min(max(wid, 1), ;
      (objBrow:nRight - objBrow:nLeft) +1)

   //  Replace modified column object.
   objBrow:setColumn(i, objCol)
next i
return nil
```

It's often not necessary to extract a column to a temporary variable just to manipulate an instance variable. Listing 25.4 is a function that will remove footings from all columns in a browse, without having to store the columns in a temporary variable.

Listing 25.4 Remove all column footings from a browse object

```
function NoFeet(objBrow)
/*
   Remove all column footings in a browse object.
*/
local i
for i := 1 to objBrow:colCount
    objBrow:getColumn(i):footing := nil
next i
return nil
```

Whoa. What is that monstrosity? Let's take it one piece at a time. The following code fragment is functionally equivalent to the single line in the FOR..NEXT loop in Listing 25.4, which should be more clear.

```
col := objBrow:getColumn(i)
col:footing := nil
objBrow:setColumn(i, col)
```

These three lines extract a column, assign NIL to the footing, and replace the old column with the modified column. However, since the getColumn() method returns an object, we can immediately reference its footing instance variable and assign it a value.

b:delColumn(nColumn)
b:insColumn(nColumn, objColumn)

As the names imply, these methods delete and insert columns in browse objects. As with b:setColumn() you must call b:configure() and then stabilize the browse window to see the effects in the browse window.

The following example illustrates the timing between column insertions and TBrowse updating the related instance variables.

```
/*
   Insert a new column between columns 2 and 3.
*/
? brow:colCount             //  5
brow:insColumn(3, objCol)
? brow:colCount             //  Still 5
brow:configure()
? brow:colCount             //  Now 6
```

The next example shows that deleting columns has the same timing issue.

```
/*
   Delete last column.
*/
? brow:colCount                 //  6
brow:delColumn(brow:colCount)
? brow:colCount                 //  Still 6
```

```
brow:configure()
? brow:colCount                    //  Now 5
```

The following example allows the user to navigate around the browse window an press the Del key to delete the current column.

```
#include "inkey.ch"

do while .t.
  do while .not. brow:stabilize()
  enddo
  key := inkey(0)
  do case
  case key == K_ESC
    exit
  case key == K_UP
    brow:up()
  case key == K_DOWN
    brow:down()
  case key == K_LEFT
    brow:left()
  case key == K_RIGHT
    brow:right()
  case key == K_DEL
    brow:delColumn(brow:colPos)
    brow:configure()
  endcase
enddo
```

b:hilite()
b:deHilite()
b:hilite() turns the browse cursor highlight on, b:deHilite() turns the highlight off. Stated more accurately, b:hilite() causes TBrowse to display the current value in the first color in the column's color pair while b:deHilite() causes the second color to be

used. Exactly which color pair is used depends on how the column was defined. See the previous discussion on b:colorSpec, b:defColor, and b:colorBlock for details, as well as the b:colorRect() method discussed below.

TBrowse reacts immediately to b:hilite() and b:deHilite(); you do not have to do any stabilization to see the effects.

If b:autoLite is true, TBrowse will automatically highlight the value where the browse cursor is currently sitting. If b:autoLite is false the browse cursor will still be accurate with respect to the row and column position within the browse window, but the current value will not be highlighted. It's up to you to call b:hilite() and b:deHilite() to manage the highlighting yourself if b:autoLite is false. The example in Listing 25.5 shows direct calls to the highlighting methods used to create a slowly flashing cursor.

Listing 25.5 A browse cursor that flashes slowly

```
function SlowFlash(browse)
/*
   A browse cursor that flashes slowly.
   On for half a second, off for a quarter second.
*/
local key
do while .t.
   brow:hilite()
   if (key := inkey(0.5)) <> 0
     exit
   endif
   brow:deHilite()
   if (key := inkey(0.25)) <> 0
     exit
   endif
enddo
return key
```

You don't have to keep turning the highlight on and off. You can turn it on and leave it on or turn it off and leave it off. That's the whole point of giving you control over the highlight.

b:colorRect(aCoords, aColors)
This method name is short for "color rectangle". It is used to establish a special color scheme for a rectangular region in the browse window. The region may include all columns but is confined to only the rows currently visible, which somewhat limits its usefulness. Still it's a handy method that allows you to quickly and easily alter row and column colors that otherwise would not be possible (or at least, not elegantly done) using the other color-related methods.

b:colorRect() takes two arrays as parameters. The first consists of the four browse window coordinates (not screen coordinates), the second is a color pair. See the discussion about b:colorSpec and b:defColor for details about color pairs.

TBrowse reacts immediately to b:colorRect(), you do not need to stabilize the window to see the results. All values within the region are displayed in the unhighlighted color, which is the first color in the color pair. When the browse cursor is moved into the region defined by b:colorRect(), the current value is displayed in the highlight color, which is the second color in the pair.

```
//  Using colors #5 and #6, display values in rows and columns
//  from row 3, column 2 through row 10, column 7.
brow:colorRect({3,2,10,7}, {5,6})
```

A comprehensive example

Is it safe to assume you didn't memorize the huge assortment of methods and instance variables just discussed? Can you picture how they all fit together at run-time? Only after considerable experimentation and practice will all these concepts come together. To demonstrate the power and versatility of TBrowse we've included a program that manages to use all of the features we've discussed in this chapter: MAXIBROW. (So named because it stands in contrast to the as-simple-as-possible MINIBROW program presented previously in the introduction to objects.) Complete source code for MAXIBROW.PRG can be found in Appendix B.

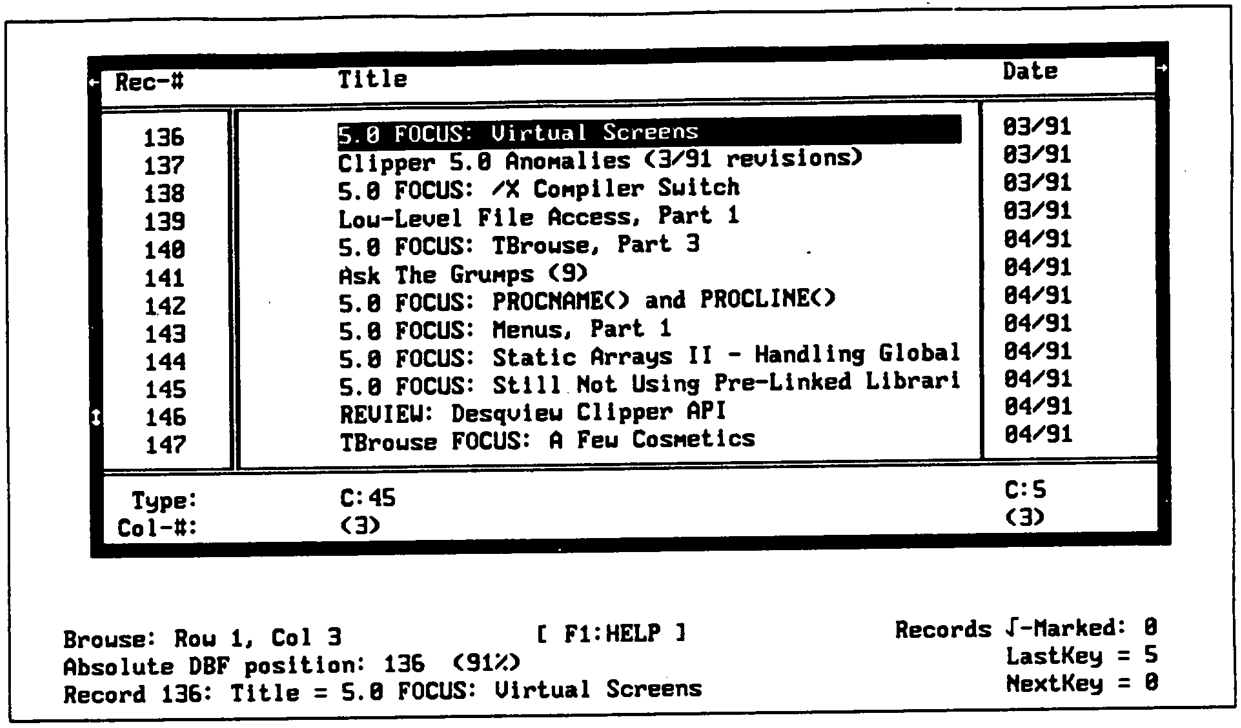
MAXIBROW is not intended to be a routine you actually implement in your applications. It goes to sometimes ridiculous lengths to incorporate all the TBrowse features. It's intended to be a TBrowse playground of sorts where you can see various instance variables and methods in action and in the context of a large application rather than in small, isolated examples. Use it to experiment — make code changes, add new features, change constants, do whatever you find interesting. The more time you spend playing with TBrowse, the better you'll understand it.

To get the most out of MAXIBROW you should have a copy of the source code handy (either printed or available at the touch of a key with a TSR editor like Borland's SideKick or even within Clipper's source-level debugger). Trace the program flow as the program executes on the screen. Use your editor's search functions to track down instance variable and method names to see how they are used.

The use of color and the formatting of the various status messages is deliberately spartan. We didn't want fancy, colorful things to get in the way of the underlying concepts. There's enough code as it is! The screen is by design very blah until you start fiddling around with the highlighting and selection functions, or supply a database with dates, negative numbers, logical values, or memo fields. Then you'll probably wish there weren't so many colors. We predict the incredibly flexible TBrowse color scheme will be responsible for a whole new generation of garish application screens.

All the features of MAXIBROW are tied to various keystrokes. The source code comments surrounding the features describe the techniques being used. The next four sections summarize what MAXIBROW can do. Figure 25.2 shows a sample screen generated by the MaxiBrow program.

Figure 25.2 Sample screen displayed by MaxiBrow.Prg



```
 Rec-#          Title                                       Date

   136          5.0 FOCUS: Virtual Screens                  03/91
   137          Clipper 5.0 Anomalies (3/91 revisions)      03/91
   138          5.0 FOCUS: /X Compiler Switch               03/91
   139          Low-Level File Access, Part 1               03/91
   140          5.0 FOCUS: TBrowse, Part 3                  04/91
   141          Ask The Grumps (9)                          04/91
   142          5.0 FOCUS: PROCNAME() and PROCLINE()        04/91
   143          5.0 FOCUS: Menus, Part 1                    04/91
   144          5.0 FOCUS: Static Arrays II - Handling Global 04/91
   145          5.0 FOCUS: Still Not Using Pre-Linked Librari 04/91
   146          REVIEW: Desqview Clipper API                04/91
   147          TBrowse FOCUS: A Few Cosmetics              04/91

  Type:         C:45                                        C:5
 Col-#:         (3)                                         (3)


Browse: Row 1, Col 3              [ F1:HELP ]       Records √-Marked: 0
Absolute DBF position: 136  (91%)                          LastKey = 5
Record 136: Title = 5.0 FOCUS: Virtual Screens             NextKey = 0
```

Navigation keys

All of the browse navigation methods are supported:

- Up, Down, Left, Right: Move one cell in any direction, will scroll rows up/down and pan columns left/right as needed.

- Home, End: Jump to first or last visible column.

- Ctrl-Home, Ctrl-End: Jump to very first or very last column.

- PageUp, PageDown: Scroll one screenful up or down.

- Ctrl-PageUp, Ctrl-PageDown: Jump to very top or very bottom of database.

- Tab, Shift-Tab: Pan the screen left or right to see more columns.

Function keys

F1	Display record numbers/columns visited during "help". Each time you press F1 the current record number is added to a list. A column-based counter is also incremented. Press F1 periodically as you move through the database. F1 also displays a brief summary of what other keystrokes do.
F2	Toggle between color and monochrome color schemes.
F3	Insert a copy of current column. Watch how changes to the original column are reflected in the copies. This is due to the referential, array-based nature of Clipper objects.
F4	Delete current column. Can't delete the last non-frozen column.
F5	Move the browse window. The Up/Down/Left/Right keys work as expected. Press any other key to finish the move. Built-in logic will prevent you from moving the window completely off the screen. Press backspace to restore coordinates to initial settings.
F6	Resize the browse window. The Up and Left keys make the window larger, Down and Right make it smaller. Press any other key to finish the resizing. Built-in logic will prevent you from making window too small to be useful. Press backspace to restore coordinates to initial settings.

F7	Rotate column positions. Non-frozen columns are shifted to the left, first column is moved to the far right. You can press F7 repeatedly to cycle through as many columns as you wish. Press Shift-F7 to rotate columns in the other direction.
F8	Toggle between drag-highlight regular navigation modes. After pressing F8 all cursor movements will enlarge a highlight box on the screen. Due to limitations inherent in TBrowse's colorRect() method the highlighted box is for the visible screen rows, only.
F9	Highlight current column. Due to aforementioned colorRect() limitation the highlight extends only for the rows currently visible.
F10	Highlight the current row. Entire row is highlighted, including columns that are not currently visible.
ESC	Finished browsing, you'll be asked to confirm that you want to exit. Press Y to exit or any other key to remain in the browse.

Editing keys

Alt-U	Toggle the global SET DELETED flag on and off.
Ctrl-U	Toggle the individual record deletion flag on and off.
Enter	Edit current cell, including a full-window editor for memo fields.
Ctrl-Enter	Clear contents of current cell and then start editing, except for memo fields.
!..chr(255)	Edit current cell, starting with the keystroke.

Other miscellaneous keys

Ctrl-Left	Make current column more narrow. Can't make column smaller than one character.
Ctrl-Right	Make current column more wide. Can't make column wider than width of original field in database.
Backspace	Clear all highlights and selections, reset all cargo instance variables, refresh entire screen. Used primarily to recover after making a mess.
Spacebar	Toggle record selection check-mark on and off. Counter at bottom of screen indicates how many have been selected. Unlike the F8, F9, and F10 highlighting functions, the check-marks are not limited to the currently visible screen. Backspace clears all selections.

Other automatic things

Other things will happen automatically depending on the contents of the database being browsed:

- Negative numbers will be in red.
- Logically false values will also be in red.
- All date columns will be in magenta, including headings and footings.
- "Thud" sounds will be heard when you attempt to scroll or pan beyond the physical boundaries of the database.

- A relative position indicator, or "elevator", will be displayed on the left side of the window if the database contains more records than can fit in the window. Important note: The indicator is maintained independent of the index, if one is being used. This is implemented very elegantly thanks to the wonderful concept of TBrowse "skipBlocks".

The source code

There's a tremendous amount of fancy browsing going on in MAXIBROW.PRG, and to help keep it all straight we've included lots of comments. Block-style comments, delimited with /* and */, are used to describe large sections of code and also are used as headers to functions.

Single line comments, //, are used when the comments are directed at the next line (or small group of lines). Two blank lines separate logical groups of source code, single blank lines separate small runs of related source code within a major group.

Other helpful conventions: If a function name is in all lowercase letters then it's part of the built-in Clipper functions. If a function name is in "proper" case it's a user-defined function and can be found in this source code file. Array names end with a trailing underscore. Manifest constants are in all upper case.

The MAXIBROW.PRG source code file contains the following functions.

- **Main(cFilename [, cIndexname])**
  Main browsing program. From DOS, send database filename and optional index filename.

- **Proper(cString)**
  Given string, returns "properized" string where first character is made uppercase. Used in this program to make the database field names look better.

YesNo(cMsg [, nSeconds])
Given a message to display an optional maximum number of seconds to wait, return .t. if "Y" or "y" is pressed, .f. if not.

HelpStat(oBrowse)
Given a browse object, use browse and current column cargo variables to display stats about how often the F1-HELP key was pressed and which records were visited.

- **FitInBox(nTop, nLeft, nBottom, nRight, aMessage)**
Draw a box at specified coordinates and display an array of message lines within the boundaries of the box. Trim the number and width of lines to fit completely inside the box.

- **RecPosition([cHow] [, nHowmany])**
This is a single function used for all three TBrowse data positioning blocks. Given the type of movement required (top, bottom or skip) and how many records to skip, function performs all the necessary database movements. If called without any parameters the function returns the current record's position within the database. The position is maintained independent of the index, if any.

- **RecDisplay(nRec, aList)**
This function is used by the record-# column data retrieval block to format the record number for display. It handles the check-mark. The current record number is passed along with an array containing record numbers that should be displayed with check-marks.

- **aCount(aX, bCountBlock [, nStart] [, nCount])**
Given an array, use the supplied code block to count the number of elements that match a condition. Optional starting element number and number of elements to evaluate. Returns count.

- **aInsert(aX, nPos [, xValue])**
  Increase size of array by inserting new value in specified position. This function combines the effects of the AINS() and ASIZE() functions.

- **aDelete(aX, nPos)**
  Decrease size of array by deleting element in specified position. This function combines the effects of the ADEL() and ASIZE() functions.

- **aRotate(aX [, lDir])**
  Rotates elements in array passed as parameter. Elements are shifted up one. First element is shifted to last element. Optional direction to shift, default is .t. and implies shift up, .f. is down. Returns nil.

- **ColumnColor(xValue)**
  Given a value of any type, returns a color-selection array based on the type and value. Used to install logic for displaying certain data types or values in special colors. For example, negative numbers in red. Any number of additional cases can be installed as you need them.

- **Navigate(oBrowse, nKey)**
  Given a browse object and a potential cursor navigation key, this function searches an internal list of keystrokes and associated browse navigation methods. If the key is found the navigation method message is sent to the browse object and the function returns .t.; if not found, function returns .f. and no action is taken.

- **EditCell(oBrowse, cFieldname, cColor)**
  This is a general-purpose browse cell content editor. It works with all database field types including memo fields. All editing occurs within the boundaries of the browse window regardless of its size or position on the screen. On exit from the cell the function passes along any browse cursor navigation that occurred.

A TBrowse toolbox

MAXIBROW.PRG provides you with examples of uses for every TBrowse and TBColumn feature. You can lift some of the source code verbatim if it fills a need, or better yet, use the code to generate ideas. Let's discuss some select portions of the MAXIBROW source code in more detail. We'll cover the following points of interest:

Browse Cursor Navigation
Color Scheme Technique
Data Source Positioning
Data Source Structure
Editing the Data Source

Browse cursor navigation

All browse cursor navigation in MAXIBROW is routed through a general-purpose function called Navigate(). The following loop forms a complete window navigation routine for any browse object.

```
#include "inkey.ch"

//  Assuming browse object "brow" has been constructed.
do while .t.
  do while (.not. brow:stabilize()) .and. (nextkey() == 0)
  enddo
  key := inkey(0)
  do case
  case Navigate(brow, key)
  case key = K_ESC
    exit
  endcase
enddo
```

Exactly which keystrokes are assigned to which browse navigation methods is up to you. Navigate() provides a handy place to store them and lends a high degree of consistency to your applications. If all browse cursor navigation is handled by Navigate() then all your browse windows will behave the same way. This is a mark of professionalism and your end-users will like the consistency.

The Navigate() function returns .t. if it can handle the key, .f. if not. This allows you to call the function when it's possible that a keystroke might be intended for browse cursor movement and then handle exceptions on your own. You can install support for as many or few keystrokes as you desire. If you want to change the behavior of a key that Navigate() handles, simply check for the key and react to it prior to calling Navigate(). Listing 25.6 contains the source code.

Listing 25.6 A general-purpose browse cursor navigation function

```
function Navigate(b, k)
/*
    Establish array of navigation keystrokes and the cursor move
    ment method to associate with each key. The array is comprised
    of two-element arrays containing the inkey() value of the key
    and a code block to execute when the key is pressed.

    This function gets passed a browse object and a potential
    navigation key. If the key is found in the array its
    associated navigation message is sent to the browse.
    Function returns .t. if navigation was handled, .f. if not.
*/
    local n

    //  Made static so it doesn't get re-initialized on every
   //  call.
    static keys_
    if keys_ == nil
      keys_ := { ;
      {K_UP,       {|| b:up()      }},;  //  Up one row
```

```
        {K_DOWN,      {|| b:down()     }},;  //  Down one row
        {K_LEFT,      {|| b:left()     }},;  //  Left one column
        {K_RIGHT,     {|| b:right()    }},;  //  Right one column
        {K_PGUP,      {|| b:pageUp()   }},;  //  Up on page
        {K_PGDN,      {|| b:pageDown()}},;   //  Down one page
        {K_CTRL_PGUP,{|| b:goTop()     }},;  //  Up to the first record
        {K_CTRL_PGDN,{|| b:goBottom()}},;    //  Down to the last record
        {K_HOME,      {|| b:home()     }},;  //  First visible column
        {K_END,       {|| b:end()      }},;  //  Last visible column
        {K_CTRL_HOME,{|| b:panHome()  }},;   //  First column
        {K_CTRL_END, {|| b:panEnd()   }},;   //  Last column
        {K_TAB,       {|| b:panRight()}},;   //  Pan to the right
        {K_SH_TAB,    {|| b:panLeft() }} ;   //  Pan to the left
        }
        endif

      //  Search for the inkey() value in the cursor movement array.
      //  If one is found, evaluate the code block associated with it.
      //  Remember these are paired in arrays: {key, block}.
      //
      n := ascan(keys_, { | pair | k == pair[1] })
      if n <> 0
        eval(keys_[n, 2])
      endif

      return (n <> 0)
```

Color scheme technique

MAXIBROW uses a fairly complex color scheme comprised of ten different color combinations. It also supports the ability to switch on the fly between color and monochrome schemes, a very handy feature when you're trying to program for a variety of monitors. This complexity is not evident in the actual source code, however, because we've taken care to separate and isolate the color configuration from the rest of the routines.

Here's a block of preprocessor #define directives that completely describe the MAXIBROW color scheme. Note how the names associated with the various components are not directly related to a particular color. They are driven by the situation and not the desired color. The colors can be changed to suit someone's whims without any adverse effects on the rest of the application. Since TBrowse always deals with colors in pairs — the currently highlighted cell is in one color, other cells in another — we've established the colors in a way that makes it clear exactly how the color scheme works.

```
/*
   Default color scheme for all columns.
   (Used with instance variable browse:colorSpec.)

   1: Regular cell
   2: Highlighted regular cell
   3: Block-selection cell
   4: Highlighted block-selection cell
   5: Checked record-#
   6: Highlighted, checked record-#
   7: Regular negative numbers and .F. values
   8: Highlighted negative numbers and .F. values
   9: Regular dates
  10: Highlighted dates


                        1    2    3    4    5    6    7    8    9    10
*/

#define COL_COLOR "W/N, N/W, W+/B, B/W, W+/G, B+/G, R+/N, W+/R, RB+/N, W+/RB"
#define COL_MONO  "W/N, N/W, N/W,  W*/N, W/N, W+/N, W+/N, N/W,  W+/N,  N/W"
```

Later, the color scheme is assigned to the browse based on the type of adapter card present at run-time.

```
useColor := iscolor()
browse:colorSpec := if(useColor, COL_COLOR, COL_MONO)
```

Although the #defines help with b:colorSpec, they don't address the actual colors, which is potentially the most confusing aspect of the way you program TBrowse colors. Color-related functions expect an array containing a color pair. If you don't assign the color pairs correctly the browse won't look and feel as logical and consistent as it should. The following preprocessor #defines, which should appear immediately following the color scheme comments and #defines listed previously, associate the color pairs with more descriptive names.

```
/*  The following make it easier to use the browse:colorSpec.
    They correspond to the color scheme defined above.
*/
#define REGULAR_CELL  {1,2}
#define BLOCKED_CELL  {3,4}
#define CHECKED_CELL  {5,6}
#define NEGVAL_CELL   {7,8}
#define DATE_CELL     {9,10}
```

Based on these #defines we can write color-related code that is immediately clear in its intent.

```
//  Date columns get special color treatment.
if type(field(n)) = "D"
  column:defColor := DATE_CELL
else
  column:defColor := REGULAR_CELL
endif

//  We want negative numbers to be in a different color.
column:colorBlock := { |n| if(n < 0, NEGVAL_CELL, REGULAR_CELL) }
//  Highlight current row.
browse:colorRect({browse:rowPos, browse:freeze +1, ;
                  browse:rowPos, browse:colCount}, ;
                  BLOCKED_CELL)
```

Data source positioning

All data source positioning in MAXIBROW is done through calls to a single function, rather than having three separate functions. Besides being easier to maintain, the use of a single function allows for some additional tricks we'll discuss in a moment. Here are the code blocks for the three data source positioning services each TBrowse object requires.

```
browse:goTopBlock    := { | | RecPosition("top")     }
browse:goBottomBlock := { | | RecPosition("bottom")  }
browse:skipBlock     := { |n| RecPosition("skip", n) }
```

Listing 25.7 contains the source code for the RecPosition() function. Note that it performs two distinct services: Record position movement and position reporting. The bulk of the code comes directly from the SkipRec() function. See Listing 25.2 for detailed source code comments.

Listing 25.7 RecPosition(), a general-purpose record positioning and status function

```
function RecPosition(how, howMany)
/*
   General-purpose record positioning function, called by TBrowse
   goTop, goBottom and skip blocks. Returns number of records
   actually skipped if in "skip" mode.

   Also can be called with no parameters to get record position
   within database, independent of presence of index.
*/
//  Assume no movement was possible
local actual := 0

static where := 1

do case
case how = "top"
   where := 1
   goto top
```

```
case how = "bottom"
   where := lastrec()
   goto bottom

case how = "skip"
   do case
      //  Moving backwards
   case howMany < 0
      do while (actual > howMany) .and. (.not. bof())
         skip -1
         if .not. bof()
           actual--
         endif
      enddo

   //  Moving forwards
   case howMany > 0
      do while (actual < howMany) .and. (.not. eof())
        skip +1
        if .not. eof()
           actual++
        endif
      enddo
   if eof()
         skip -1
      endif

   //  No movement requested, re-read current record
   otherwise
        skip 0
   endcase

  //  No parameters passed, return current position.
otherwise
    return where
endcase

//  Update position tracker and prevent boundary wrap.
where += actual
where := min(max(where, 1), lastrec())

return actual
```

One major advantage of this technique is that it keeps all data source positioning activities in one place. We'll take advantage of this later when we see how easy it is to browse a subset of a data source without resulting to a slow and inefficient SET FILTER function.

More immediately, we see that we can maintain a static data source position variable, called "where" in the function's source code. This variable starts off with a value of one, meaning we're positioned at the top of the file. As RecPosition() gets requests for record movements, we can update the variable with the new position. If we are not using an index file, all we've accomplished is a very convoluted way to determine the record number. However, this technique works just as well with an index active, and that's the reason it's being used. A call to RecPosition() with no parameters causes it to return the current value of the position tracking variable. The main browsing routine in MAXIBROW displays the current record number at the bottom of the screen along with an the absolute position. Based on comparing the position with the total record count we can display an indicator in the browse window that helps the user keep track of the relative position in the database.

```
// Before getting started, initialize a variable used
// as local copy of the position indicator. This is done
// once, outside of the main browsing loop.
local relPos := 1

do while .t.
  //
  // Update relative position indicator, but only if there
  // are more records in database than can fit on the screen.
  //
  if lastrec() > browse:rowCount

    // Erase previous indicator by re-displaying
    // the section of the left edge of the window.
    // (Adjust the +2 to accommodate more than a single
    // row of column headings.)
    @ browse:nTop +2 +relPos, browse:nLeft -1 say chr(219)
    //  Calculate a new screen row to display the indicator,
    //  based on a percentage of the width of the window.
    relPos := min((RecPosition()/lastrec()) *browse:rowCount, ;
                  browse:rowCount -1)

    //  Save the current color, invert it, display the
    //  indicator character, then restore the original color.
    clr := setcolor("I")
    @ browse:nTop +2 +relPos, browse:nLeft -1 say chr(18)
    setcolor(clr)
  endif
  //
  //  The rest of your browse routine...
  //
enddo
```

There are several tricks present in this technique. First, we've taken care of the situation where no indicator is needed by skipping the whole process if we can already see all the records. An indicator is nice but not if it confuses the user.

Second, we update the indicator by first replacing its current position with the hunk of window border that it obscures, calculating a new position, and displaying the indicator again. Sometimes the indicator may stay right where it is if the database is so huge that it takes several screenfuls of data to make it budge.

Third, calculating where exactly to display the indicator is actually quite simple. Dividing the current position by the record count gives you a percentage to apply against the number of screen rows visible in the browse window. Suppose there are 600 records in the database and ten rows to display them in. If we're positioned at the 200th record, the indicator should be placed 33% of the way down the screen, starting from the first row of the browse window. 33% of ten lines is a hair over 3 lines, so we move the indicator down three rows. This is accurate enough for the purpose served by the indicator.

A final trick is very handy in situations where you want something to be in a different but compatible color than its surroundings. Setting the color to "I" does exactly that — it swaps the current foreground and background colors. If you were previously in "BG+/W", setting the color to "I" yields "W/BG+". The letter I is for inverse, it's not really a color but a color operator of sorts.

Let's step back for a moment and cover a limitation inherent in this technique: You must route 100% of the data source positioning through the RecDisplay() function. If you move the record pointer outside of the function the "where" variable won't get updated and the indicator will stray. This means you can't jump to a record via GOTO or SEEK; only the regular browse navigation methods are supported. Aside from that, you can use RecPosition() and a relative position indicator to spruce up any browse window.

Data source structure

If you read the source code carefully you may notice that we've gone to great lengths to load an array with database structure information and then maintain that array as the browse and column objects go through various gyrations. This was done in an attempt to make MAXIBROW, and any browse routines you create based on this design, more independent of the structure of the actual data source. If you were browsing an array with two dimensions, very little of the MAXIBROW source code needs to be changed. Follow these steps.

1. Build a stru_ array that has the same general structure as the array returned by DBSTRUCT() for database structures. (See Chapter 9, "Arrays," for details on how to manipulate arrays with multiple dimensions.) That is, the stru_ array should describe the array containing the data you wish to browse — names for the fields, data types, widths and so on.

2. Create c:block column blocks that reference array elements rather than database field names.

3. Create b:goTopBlock, b:goBottomBlock, and c:skipBlock functions that operate on arrays rather than databases.

And that is all there is to it. Once MAXIBROW is up and running almost all of its features will work with arrays. Of course, arrays don't have record numbers, deleted flags, and such so those features won't translate automatically (although all are possible with additional editing).

See the section on Browsing Arrays, later in this chapter, for details on how to perform steps 1 and 2.

Editing the data source

Allowing the user to scroll through a database with a tool as wonderful as TBrowse is sometimes a tease — you can look but you can't touch! Fortunately it's a simple matter to provide complete editing capabilities to any TBrowse window. MAXIBROW contains a complete function, called EditCell(), that handles the entire process. It even pops up a special editor for memo fields. You'll be surprised by the relatively small amount of source code it takes to pull this off. Here's what a call to EditCell() looks like within the body of the MAXIBROW browsing loop. The cell editor gets invoked whenever the user presses the Enter or Ctrl-Enter keys, or when a key within the normal keyboard range is pressed.

```
case (key == K_ENTER) ;         //  Open current cell for editing.
.or. (key == K_CTRL_ENTER) ;    //  Clear cell contents and edit.
.or. (key > K_SPACE)            //  Edit by starting to type.
//
 EditCell(browse, ;
         stru_[browse:colPos -1, DBS_NAME], ;  //  Field name
         EDIT_COLOR)
```

The simplicity of the call to EditCell() is obscured by MAXIBROW's need to be completely generic so it can be used with any database structure. It also uses a manifest constant for the color string, defined previously (see the earlier discussion on the MAXIBROW color scheme). Here's the call written as it would look for editing the LastName field in bright white letters on a blue background.

```
EditCell(browse, "LastName", "B+/W")
```

That's all there is to it. The browse object carries with it everything we need to know to perform the edit. You may be wondering, doesn't the browse object know about the field name as well? It does, but there's no guarantee that we'll be able to pull the actual field name out of it. The column's data retrieval block is a code block and we can't reverse-engineer the field name. We could use the column's cargo variable to store the field name but it's just as easy to send it as a parameter since we have to know it to create the column, anyway.

One other important point to be aware of before you implement this function. The column field blocks must be capable of storing values back to the data source, not just retrieving values. The edit will appear to work just fine, but nothing will actually be saved. Here are three data retrieval blocks. The first won't work because it doesn't have the ability to store values back to the data source. The next two do because FIELDBLOCK() and it's cousin, FIELDWBLOCK(), return the proper kind of code block, and the way the third one is defined gives it the same capability.

```
//  Won't work if data storage is needed.
col1:block := { || CUST->FirstName }

//  These both will work.
col2:block := fieldblock("LastName")
col3:block := { |x| if(x == nil, Phone, Phone := x) }
```

The third example works the way it does because whenever Clipper wants to store something to such a code block it passes a parameter when EVALuating it. If no parameter is passed it means Clipper only wants the current value. Complete source code for EditCell() is found in Listing 25.8.

Listing 25.8 EditCell(), a general-purpose browse cell editing function

```
function EditCell(b, fieldName, editColor)
/*
   General-purpose browse cell editing function, can handle all
   database field types including memo fields. If you want the
   edits to "stick" you must assign fieldblock()-style
   column:block instance variables. All editing, including memo-
   edit, is done within the boundaries of the browse window. On
   exit, any appropriate browse cursor navigation messages are
   passed along.
   Note: In order to browse a memo field the column heading must
   be defined. This function uses the heading to display a mes-
   sage.
*/
#include "inkey.ch"

#define K_SPACE       32
#define K_CTRL_ENTER  10
#define lstr(n)       ltrim(str(n))

local c, k, clr, crs, rex, block, cell

//  Retrieve the column object for the current cell.
c := b:getColumn(b:colPos)

//  Create a field block used to check for a memo field
//  and later used to store the edited memo back. It's
//  done this way so you can have the browse window display
//  a notation like "memo" rather than displaying a small
//  hunk of the real memo field.
//
block := fieldblock(fieldName)
```

```
  //  Can't just "get" a memo, need a memoedit.
  if valtype(eval(block)) == "M"

    //  Tell the user what's going on.
    @ b:nTop, b:nLeft clear to b:nBottom, b:nRight
    @ b:nTop, b:nLeft say ;
      padc("Memo Edit: Record " +lstr(recno()) ;
          +', "'+ c:heading +'" Field', b:nRight -b:nLeft)
    @ row() +1, b:nLeft say replicate("-", b:nRight -b:nLeft +1)

    //  Turn cursor on and perform the memo edit
    //  using the specified color.
    crs := setcursor(1)
    clr := setcolor(editColor)
    cell := memoedit(eval(block), b:nTop +2, b:nLeft, b:nBottom,
            b:nRight)
    setcursor(crs)
    setcolor(clr)

    //  If they didn't abandon the edit, save changes.
    //  When passed a parameter, fieldblock-style code
    //  blocks store the value back to the database.
    //  Handiest darn thing they ever stuck in this language.
    if lastkey() <> K_ESC
      eval(block, cell)
    endif

    //  We mussed up the entire window, tell TBrowse to clean it
    // up.
    b:configure()

//  Regular data type, do a GET/READ.
else

  //  Pass along any additional keystrokes.
  if lastkey() == K_CTRL_ENTER
    keyboard(chr(K_CTRL_Y))
  elseif (lastkey() > K_SPACE) .and. (lastkey() < 256)
    keyboard(chr(lastkey()))
  endif
```

```
    //  Create a get object for the field.
    cell := getnew(row(), col(), ;
                   c:block, fieldName,, "W/N,"+editColor)

    //  Allow up/down to exit the read, and turn the cursor off.
    rex := readexit(.t.)
    crs := setcursor(1)

    //  Perform the read.
    readmodal({cell})

    //  Restore original cursor and read-exit states.
    setcursor(crs)
    readexit(rex)

    //  If user hit a navigation key to exit, do it.
    if Navigate(b, lastkey())

    //  If they pressed Enter, advance to next column.
    elseif lastkey() = K_ENTER
      if b:colPos < b:colCount
        b:right()
      else
        b:down()
        b:colPos := b:freeze +1
      endif
    endif

    //  We changed the field value and TBrowse doesn't know it.
    //  So we must force a re-read for the current row.
    b:refreshCurrent()
  endif

return nil
```

EditCell() first distinguishes between memo fields and other data types. Memo fields are processed with a call to the MEMOEDIT() function while all other data types use the GETSYS system for @..GET-style editing. EditCell() can be implemented in any browse routine, triggered by any keystroke you desire.

If working with something other than a memo field, EditCell() next fetches the last keystroke to see how it should react to the user. Pressing Enter simply causes the current cell to be opened for editing. Ctrl-Enter first clears the field by stuffing a Ctrl-Y into the keyboard buffer. Keys within the normal keyboard range are placed back into the keyboard buffer so they are treated as part of the cell edit. This allows the user to simply start typing when they want to enter data, rather than having to remember to first type a special "I want to edit" keystroke. You don't have to give the user such easy access to the data, sometimes it's preferable to require a special keystroke.

TBrowse versus SEEK and GOTO

If you move the record pointer directly, instead of indirectly through the data source positioning blocks, you may find that TBrowse does not maintain the row position correctly. This is not due to an error in TBrowse but rather a lack of communication between TBrowse's internal tracking of the record position and your direct manipulation.

Also troublesome at times is not always staying on the same row following a call to b:refreshAll(). Fortunately there's a simple solution to both problems in Listing 25.9.

Listing 25.9 A replacement for b:refreshAll() which keeps the current row highlighted

```
function dbRefresh(browse)
/*
   A replacement for b:refreshAll() that will keep the
   current row highlighted, used primarily when you've
   moved the record pointer via SEEK or GOTO such that
   the record is still on the screen. If the entire screen
   has to be redrawn with new data, this function will
   have no net effect beyond the actual b:refreshAll()
   and won't cause any delays.
*/
```

```
//  Remember where we ended up.
local rec := recno()

  //  Perform the initial stabilizing process.
  browse:refreshAll()
  do while .not. browse:stabilize() ; enddo

  //  Most of the time this will be false,
  //  so no work will be done.
  do while recno() <> rec
    browse:up()
    //  Need to stabilize again.
    do while .not. browse:stabilize() ; enddo
  enddo

return nil
```

Here are some examples of how dbRefresh() is used.

```
goto marked_rec
VENDOR->(dbRefresh(brow))
seek someValue
ITEMS->(dbRefresh(brow))
```

Browsing within a WHILE condition

Sometimes all of the records in a database are not relevant for the task at hand, for example, viewing product orders placed by a single customer. A simple solution is the SET FILTER command. It eliminates from consideration all records not matching a condition.

```
select ORDERS
set filter to ORDERS->Customer == CUSTOMER->ID
goto top
//
//  Display a browse window.
//  ...
```

This is an acceptable solution only when dealing with a database with a small number of records, or when a majority of the records match the filter condition. Problems surface when the GOTO TOP operation has to evaluate hundreds or even thousands of records to find the first one that satisfies the condition. And for the very same reason, pressing PageDown may send the filter on a fruitless search for another screenful of data even if there isn't any. The filter will continue to evaluate records until it hits the end of the database, even if it means the user has to wait five minutes or more! Clearly, there must be a better way.

A preferable way to handle the situation is to have an index key based on the condition we want to browse. For the customer orders example it means having an index on at least the customer ID for each order. More than likely the database already has such an index as an aid in creating and maintaining the data.

```
//  Create an index on customer ID and purchase order number.
select ORDERS
index on Customer +PO to ORDCUST
```

Having TBrowse operate on a single customer's orders is a simple matter of adding some additional logic to the RecPosition() function listed previously. We'll need to pass the function two additional parameters: the value of the first index key that matches the condition and a code block that evaluates a record and returns .t. if it matches the condition.

Let's do the easiest ones first: the GOTO TOP and GOTO BOTTOM functions needed for servicing the b:goTop() and b:goBottom() navigation methods. Since we've got an index, it's trivial to go to the first record that matches our condition, just SEEK it.

```
//  GOTO TOP
seek firstKey
```

At this point the database is either pointing to the first record that matches the condition, or to end-of-file. We can handle the EOF() situation easily enough on our own before trying to browse the data so there's no reason to make it more complex than it needs to be in RecPosition().

GOTO BOTTOM is more troublesome. How do we quickly locate the last record that matches the condition? A brute-force method is simply skipping through records that match the condition until you find one that doesn't. Back up one record and there you have it. While this is certainly an improvement over the SET FILTER command's need to run through the entire database, it still requires that potentially thousands upon thousands of records be processed. Hardly a productive use of your user's time.

```
// GOTO BOTTOM
lastKey := left(firstKey, len(firstKey) -1) +chr(255)
set softseek on
seek lastKey
skip -1
```

This relies on the fact that with SOFTSEEK on, the SEEK command will find the index key value that either matches or is greater than the one we're looking for. We constructed a key value that differs only minutely from the correct key by changing the last character to a chr(255), which is a value that is guaranteed to be sorted last in the index. It's very unlikely that chr(255) exists in your database files, or if it does, even more unlikely that it's part of an index key. Since SOFTSEEK makes SEEK find the first key greater than the one we specify, we'll end up positioned one record past the range we are interested in, so we back up one record.

In practice, we'll need to save the current status of SET SOFTSEEK and restore it when we're done. A general-purpose function must not have any side effects to cause problems in other parts of the application.

All that's left is the SKIP service. There are no tricks involved, just some additional conditions in the logic. While skipping forward or backward we have to check to see that we haven't left the range of records that match the condition. That's where the code block parameter comes in. When evaluated it should return .t. only when the record in question is within the range of records we wish to see. Here's an example of the code block needed in our customer order example.

```
condition := { |cust| ORDERS->Customer == cust }
```

The "skip forward" portion of the RecPosition() function then looks like the following.

```
do while (actual < howMany) .and. (.not. eof()) ;
        .and. eval(condition, firstKey)
  skip +1
  if (.not. eof()) .and. eval(condition, firstKey)
    actual++
  endif
enddo
if eof() .or. (.not. eval(condition, firstKey))
  skip -1
endif
```

Skipping backwards requires the exact same kind of logic. See Listing 25.10 for the complete function.

Listing 25.10 Record positioning function for browsing with a while condition

```
function PosWhile(how, firstKey, condition, howMany)
/*
   General-purpose record positioning function, called by
   TBrowse goTop, goBottom and skip blocks. Returns number
   of records actually skipped if in "skip" mode.

   Note: It's assumed the database is already positioned at
   the first matching key (for example, after a SEEK).
*/
```

```
// Assume no movement was possible
local actual := 0
local softStat

do case
case how == "top"
  seek firstKey

case how == "bottom"
  softStat := set(_SET_SOFTSEEK, .t.)
  seek (left(firstKey, len(firstKey) -1) +chr(255))
  skip -1
  set(_SET_SOFTSEEK, softStat)

case how == "skip"
    do case
    // Moving backwards
    case howMany < 0
      do while (actual > howMany) .and. (.not. bof()) ;
               .and. eval(condition, firstKey)
       skip -1
       if (.not. bof()) .and. eval(condition, firstKey)
         actual--
       endif
    enddo
    if (.not. eval(condition, firstKey))
      skip +1
    endif

    // Moving forwards
    case howMany > 0
      do while (actual < howMany) .and. (.not. eof()) ;
               .and. eval(condition, firstKey)
       skip +1
       if (.not. eof()) .and. eval(condition, firstKey)
         actual++
       endif
    enddo
    if eof() .or. (.not. eval(condition, firstKey))
      skip -1
    endif

    //  No movement requested, re-read current record
    otherwise
      skip 0
    endcase
  endcase
return actual
```

To use PosWhile() in your browse operations you need only assign values to the two additional parameters.

```
//  WhichCust is assigned by the user.
//
whichCust := AskWhichOne()

browse:goTopBlock    := { | | PosWhile("top",    whichCust) }
browse:goBottomBlock := { | | PosWhile("bottom", whichCust) }
browse:skipBlock     := { |n| PosWhile("skip",   whichCust, ;
                          { |x| ORDERS->Customer == x }, n) }
```

All three positioning blocks need to know the value of the first index key. Only the GOTO BOTTOM block has to be concerned with the "while" condition.

Keep in mind that if you alter the value of an index key you'll need to issue a b:refreshAll() to make certain TBrowse sees the change.

Browsing arrays

TBrowse can be used to view any form of data that occurs in predictable rows and columns. It is not limited to database browsing in any way. This flexibility comes from the very same browse instance variables that allow us to browse for a WHILE condition as discussed in the preceding section: b:goTopBlock, b:goBottomBlock, and b:skipBlock. All you need to do is supply code blocks that perform these services and TBrowse will never know the difference.

With this in mind we can use TBrowse to view the contents of arrays. We'll take a look at two varieties: arrays with one dimension and symmetrical arrays with two dimensions. By symmetrical we mean arrays that have a consistent and predictable structure. Examples of symmetrical arrays with two dimensions are the arrays returned by the DIRECTORY() and DBSTRUCT() functions. We'll write TBrowse routines to view both of these arrays. But first, let's start with something easier.

Arrays with one dimension

The easiest kind of arrays to browse are those with only a single dimension. The first order of business is quickly building an array to browse, and a list of files in the current subdirectory can be made into an array in a few simple lines of code. (See Chapter 9, "Arrays," for a detailed discussion of arrays and the DIRECTORY() function).

```
//  Load the files_ array with filenames
//  found in current subdirectory.
files_ := {}
aeval(directory(), { |f_| aadd(files_, f_[1]) } )
```

The TBrowse data source positioning blocks want to be able to send us to the top and bottom of the array. An array does not have a built-in pointer like database files do, so we'll have to supply one.

```
//  This will be our "element pointer".
element := 1
```

Setting up a browse object is straightforward. Just create a browse object and a single column object since the array has only one dimension.

```
//  General browse setup.
list := TBrowseNew(10, 30, 18, 50)
list:headSep := "-"
list:footSep := "="
```

```
//  The file name column.
filename := TBColumnNew("File Name", { || files_[element] } )
```

Now we need some data positioning code blocks. Based on our element pointer we can reference the first and last elements very easily.

```
list:goTopBlock    := { || element := 1 }
list:goBottomBlock := { || element := len(files_) }
```

Skipping forwards and backwards for b:skipBlock would be just as easy were it not for the fact that TBrowse will often request that we skip further than the limits of the array. This in itself is not a problem because TBrowse doesn't expect every single position request to succeed completely. It does demand, however, that we inform it how many elements we were able to skip so it can adjust the display accordingly. Listing 25.11 is a function that will do the job. You pass it the array, current element position and how many elements to skip.

Listing 25.11 ArraySkip(), a b:skipBlock function for arrays

```
function ArraySkip(aLen, curPos, howMany)
/*
   General purpose array skipping function intended for
   use with TBrowse b:skipBlocks. You must pass the curPos
   parameter by reference (use the @ operator) for this
   function to work correctly with TBrowse.
*/
local actual

if howMany >= 0                  // Moving forward?
  if (curPos +howMany) > aLen // Can't go that far!
    actual := aLen - curPos   // Actual is whatever is left
    curPos := aLen            // Put pointer at end
   else                        // Can move the whole distance...
     actual := howMany         // Actual is number requested
     curPos += howMany         // Move pointer forward
   endif
 else                          // Moving backward?
   if (curPos +howMany) < 1    // Can't go that far!
     actual := 1 -curPos       // Actual is whatever was left
     curPos := 1               // Put pointer at top
   else                        // Can move the whole distance...
     actual := howMany         // Actual is number requested
     curPos += howMany         // Move pointer backward
   endif
 endif
 return actual
```

Used in our file listing, example the b:skipBlock looks like this:

```
list:skipBlock := { |n| ArraySkip(len(files_), @element, n) }
```

The "pass by reference" symbol, @, is required because we want the ArraySkip() function to alter the value of the element pointer to our routine and return the actual skip count to TBrowse. The only way to do both in the same function is to pass a reference to the element pointer so ArraySkip() can alter it.

At this point we have the browse and column objects ready to go, so let's combine them and get the window up on the screen. We'll call upon MAXIBROW's Navigate() function (discussed earlier in this chapter) to provide a set of browse navigation keys.

```
//  Add the file name column object to the browse object.
list:addColumn(filename)

//  Display the window and process navigation keystrokes.
do while .t.
  do while .not. list:stabilize()
  enddo
  if .not. Navigate(list, inkey(0))
```

```
      exit
    endif
  enddo
```

Listing 25.12 contains source code for the complete file name browsing routine, including the housekeeping code we omitted for clarity in the previous examples.

Listing 25.12 Browsing an array of file names

```
function BrowArray()
/*
   Load an array of file names and browse it.
*/
local files_, element, list, key

  //  Load the files_ array with filenames
  //  found in current subdirectory.
  files_ := {}
  aeval(directory(), { |f_| aadd(files_, f_[1]) } )

  //  This will be our "element pointer".
  element := 1

  //  General browse setup.
  list := TBrowseNew(10, 30, 18, 50)
  list:headSep := "-"
  list:footSep := "="

  //  Assign the positioning blocks.
  list:goTopBlock    := { || element := 1 }
  list:goBottomBlock := { || element := len(files_) }
  list:skipBlock := { |n| ArraySkip(len(files_), @element, n) }

  //  Add a file name column object to the browse object.
  list:addColumn(TBColumnNew("File Name", ;
                 { || padr(files_[element], 12) } ))

  //  Display the window and process navigation keystrokes.
  do while .t.
```

```
    do while .not. list:stabilize()
    enddo
    if .not. Navigate(list, inkey(0))
      exit
    endif
  enddo
return nil
```

The thing to keep in mind about this example is not the few things that are different about browsing an array, but that most of the source code would be exactly the same if the file listing was stored in a database instead of an array. Everything you've learned so far about browse and column objects is applicable to arrays.

Arrays with two dimensions

Browsing an array with two dimensions is done the same way as with one dimension, the only exception being the column object data retrieval blocks must refer to the additional dimension in order to extract values from the array. Let's return to the previous example and browse a complete set of directory information instead of just the file names. Compare Listing 25.12 with 25.13 to see how few changes are needed to support the additional dimension.

Listing 25.13 Browsing an array of directory information

```
function BrowArray2()

local dir_, element, list, key

  //  Load the dir_ array with a complete set of
  //  directory information for the current subdirectory.
  dir_ := directory()

  // This will be our "element pointer".
  element := 1
  //  General browse setup.
  @ 9, 19 to 20, 61 double
  list := TBrowseNew(10, 20, 20, 60)
  list:headSep := "═╤═"
  list:colSep  := " │ "
  list:footSep := "═╧═"

  //  Assign the positioning blocks.
  list:goTopBlock    := { || element := 1 }
  list:goBottomBlock := { || element := len(dir_) }
  list:skipBlock := { |n| ArraySkip(len(dir_), @element, n) }

  //  File name is first element in array.
  list:addColumn(TBColumnNew("File Name", ;
       { || padr(dir_[element, 1], 12) } ))

  //  Size is second element.
  list:addColumn(TBColumnNew("Size", ;
       { || transform(dir_[element, 2], "999,999,999") } ))

  //  Date and time are third and fourth elements.
  //  We'll skip the file attributes in the fifth element.
  list:addColumn(TBColumnNew("Date", { || dir_[element, 3] } ))
  list:addColumn(TBColumnNew("Time", { || dir_[element, 4] } ))

  //  Display the window and process navigation keystrokes.
  do while .t.
    do while .not. list:stabilize()
    enddo
    if .not. Navigate(list, inkey(0))
      exit
    endif
  enddo
return nil
```

Browsing text files

Browsing the contents of text files is as easy as browsing arrays providing you have enough memory to load the file. Once in memory we can use Clipper memo processing functions to do most of the work for us.

MEMOREAD()	Read the file into memory.
MLCOUNT()	Count the number of lines.
MEMOLINE()	Return the specified line number.

Let's use the MEMOREAD() function to load a file into a memory variable.

```
//  Load the contents of the specified file.
textfile := memoread("TEST.TXT")
```

You can think of the text file as an array with one dimension. The length of the file in this sense is the number of lines it contains.

```
//  Determine how many lines are in the file.
lastLine := mlcount(textfile)
```

As with arrays we also need to maintain a pointer that keeps track of our position in the file.

```
//  A pointer used to keep track of which line we're on.
line := 1
```

Creating the browse and column objects should look familiar. The only wrinkle is the use of the memoline() function to retrieve a line of text for TBrowse to display.

```
txt := TBrowseNew(10, 20, 18, 60)
txt:addColumn(TBColumnNew( , ;
    { || memoline(textfile, 254, line)} ))
```

We've used a line length of 254, which is the maximum MEMOLINE() can handle. This is done so that the lines in the text file do not wrap to fit the window, which could make some files very difficult to read. Other files, however, might benefit from the automatic line wrapping. Here's an alternate way to call MEMOLINE() such that the text will wrap to fit the browse window.

```
txt:addColumn(TBColumnNew( , ;
    { || memoline(textfile, txt:nRight -txt:nLeft, line)} ))
```

As with arrays, the tricky part is establishing the data source positioning code blocks. The way text file lines are organized is so similar to that of arrays that we can use the ArraySkip() function found in Listing 25.10.

```
//  The data positioning blocks.
browse:goTopBlock     := { || line := 1 }
browse:goBottomBlock  :=   { || line := lastLine }
browse:skipBlock      :=   { |n| ArraySkip(lastLine, @line, n) }
```

Since you've seen the rest of the technique so many times by now we'll dispense with the details and just list the final version of the function in Listing 25.14.

Listing 25.14 A file browser for small text files

```
function BrowText(filename)
/*
   A general-purpose file browser for small text files.
   Pass a file name from DOS.
*/
local textfile, lastLine, line, txt

   if filename == nil
     ? "Must specify a file name."
```

```
    return nil
  endif

  //  Load the contents of the specified file.
  textfile := memoread(filename)

  //  Determine how many lines are in the file.
  lastLine := mlcount(textfile)

  //  A pointer used to keep track of which line we're on.
  line := 1

  @ 8, 19 say "File: " +filename
  @ 9, 19 to 21, 61

  //  Create the browse object.
  txt := TBrowseNew(10, 20, 20, 60)

  //  Add a column that displays a line of text.
  txt:addColumn(TBColumnNew(, ;
    { || memoline(textfile, 254, line)} ))

  //  The data positioning blocks.
  txt:goTopBlock     := { || line := 1 }
  txt:goBottomBlock  := { || line := lastLine }
  txt:skipBlock := { |n| ArraySkip(lastLine, @line, n) }

  //  Display the window and process navigation keystrokes.
  do while .t.
    do while .not. txt:stabilize()
    enddo
    if .not. Navigate(txt, inkey(0))
      exit
    endif
  enddo
return nil
```

Multiple simultaneous browses

Throughout this chapter we've always been working with a single browse object at a time. While it's debatable whether or not there's a conceptual limit on how many browses we want to consider at a time, there's no practical limit. All of the browse methods and instance variables we've discussed can be applied to any arbitrary browse object, the same way a single set of column instance variables can be applied to any number of columns.

As a final example in this chapter, let's close with a function that browses a list of database files in one window and their structures in another. Both windows will be on the screen, processing navigation keystrokes and updating their respective windows, simultaneously — see Listing 25.15 Does this sound like a juggler's final, audience-dazzling trick before the curtain falls? Start spinning the plates and a let's have a drum roll, please!

Listing 25.15 Two simultaneous browse windows, one with a listing of database file names and the other with corresponding field structures

```
function DBFdirect(fileSpec)
/*
   Call this function from DOS. It will display
   two browse windows, one with database file name
   information and one with the field structure of
   the currently highlighted database..

   If you think this routine would form an excellent
   base for a general purpose database utility and are
   already dreaming up specifications and features...
   you've been bitten by that Clipper 5 bug and we
   wish you well! Turn off the lights when you leave
   and try to get some sleep once in a while.
*/
#include "inkey.ch"
#include "directry.ch"
#include "dbstruct.ch"

local dir_, stru_
local dbf, dirPos
local stru, struPos
local i, bytes, col, key

  //  Default filespec if none specified.
  if fileSpec = nil
    fileSpec := "*.dbf"
  elseif .not. ("." $ fileSpec)
    fileSpec += ".dbf"
  endif

  setcursor(0)
  @ 0,0 clear
  @ 1,0 say padc("Database Directory & Structure Viewer: " ;
                +fileSpec, 80)
  @ 2,0 say "Loading..."

  //  Load directory entries.
  dir_ := directory(fileSpec)
  if len(dir_) = 0
    @ 2,0
    @ 2,0 say "No files found."
    return nil
  endif

  //  This array will hold database structures.
  stru_ := {}

  //  For each database file in the directory...
  for i := 1 to len(dir_)
    @ 2,11 say i
    use (dir_[i, DBS_NAME]) shared

    //  Add record and field counts to the directory array.
    aadd(dir_[i], lastrec())
    aadd(dir_[i], fcount())

    //  Add current database's structure to the array.
    aadd(stru_, DBSTRUCT())
    use
```

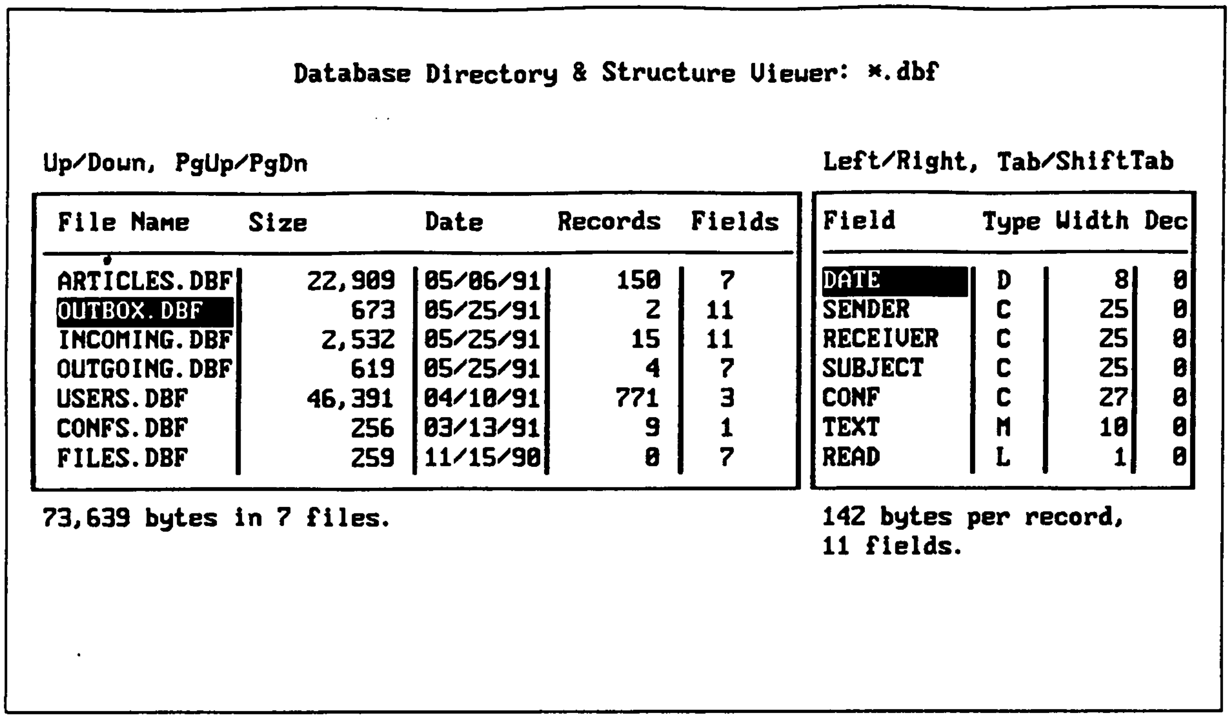
```
next i

//  Sum the file sizes.
bytes := 0
aeval(dir_, { |f_| bytes += f_[F_SIZE] } )

//  Clear the "loading" message.
@ 2,0

//  Draw the static portions of the screen.
@  4,1 say "Up/Down, PgUp/PgDn"
@  5,0 to 15,52
@ 16,1 say ltrim(transform(bytes, "999,999,999")) ;
         +" bytes in " ;
         +ltrim(str(len(dir_))) +" files."

@ 4,54 say "Left/Right, Tab/ShiftTab"
@ 5,53 to 15,79

//  Construct a browse object for the directory array.
//
dirPos := 1
dbf := TBrowseNew(6,1,14,51)
dbf:headSep := "═╤═"
dbf:colSep  := " │ "
dbf:goTopBlock    := { | | dirPos := 1 }
dbf:goBottomBlock := { | | dirPos := len(dir_) }
dbf:skipBlock     := { |n| ArraySkip(len(dir_), @dirPos, n) }

//  Directory window columns.
dbf:addcolumn(TBColumnNew("File Name", ;
    { || padr(dir_[dirPos, F_NAME], 12) } ))
dbf:addcolumn(TBColumnNew("Size",      ;
    { || transform(dir_[dirPos, F_SIZE], "99,999,999 ") } ))
dbf:addcolumn(TBColumnNew("Date",      ;
    { || dir_[dirPos, F_DATE] } ))
dbf:addcolumn(TBColumnNew("Records",   ;
    { || transform(dir_[dirPos, 6], "999,999 ") } ))
dbf:addcolumn(TBColumnNew("Fields",    ;
    { || transform(dir_[dirPos, 7], "999") } ))
```

```
// Construct a browse object for the structure array.
//
struPos := 1
stru := TBrowseNew(6,54,14,78)
stru:headSep := "¬-"
stru:colSep  := "|"
stru:goTopBlock    := { | | struPos := 1 }
stru:goBottomBlock := { | | struPos := len(stru_) }
stru:skipBlock     := ;
     { |n| ArraySkip(len(stru_[dirPos]), @struPos, n) }

// Structure window columns.
stru:addcolumn(TBColumnNew("Field", ;
     { || padr(stru_[dirPos, struPos, DBS_NAME], 10) } ))
stru:addcolumn(TBColumnNew("Type",  ;
     { || padc(stru_[dirPos, struPos, DBS_TYPE],  4) } ))
stru:addcolumn(TBColumnNew("Width", ;
     { || str(stru_[dirPos, struPos,  DBS_LEN],   5) } ))
stru:addcolumn(TBColumnNew("Dec",   ;
     { || str(stru_[dirPos, struPos,  DBS_DEC],   3) } ))

do while .t.

  // If the directory window position was altered,
  // we must update the structure window to match it.
  if .not. dbf:stable
    stru:rowPos := 1
    stru:goTop()
    stru:refreshAll()
  endif

  // Stabilize both windows.
  do while (.not.. dbf:stabilize()) .and. (nextkey() == 0)
  enddo
  do while (.not. stru:stabilize()) .and. (nextkey() == 0)
  enddo

  // Update the structure summary lines.
  @ 16,54
```

```
    @ 17,54
    bytes := 0
    aeval(stru_[dirPos], { |s_| bytes += s_[DBS_LEN] } )
    @ 16,54 say ltrim(str(bytes)) +" bytes per record."
    @ 17,54 say ltrim(str(len(stru_[dirPos]))) +" fields."

    key := inkey(0)

    //  Database window navigation keys
    do case
    case key == K_UP
      dbf:up()
    case key == K_DOWN
      dbf:down()
    case key == K_PGUP
      dbf:pageUp()
    case key == K_PGDN
      dbf:pageDown()

    //  Structure window navigation keys.
    case key == K_LEFT
      stru:up()
    case key == K_RIGHT
      stru:down()
    case key == K_TAB
      stru:pageDown()
    case key == K_SH_TAB
      stru:pageUp()
    case key == K_ESC
      exit
    endcase
  enddo
  setcursor(1)
return nil
```

Try this out in a directory full of dozens of database files with large numbers of fields. You'll be amazed how well TBrowse can handle such things. Figure 25.2 shows a sample screen generated by DBFdirect().

Figure 25.2. Typical display from DBFdirect()



```
                 Database Directory & Structure Vieuer: *.dbf

Up/Doun, PgUp/PgDn                                         Left/Right, Tab/ShiftTab
┌──────────────────────────────────────────────────────┐  ┌──────────────────────────┐
│ File Name    Size        Date       Records  Fields  │  │Field     Type Width Dec  │
│──────────────────────────────────────────────────────│  │──────────────────────────│
│ ARTICLES.DBF│  22,909 │05/06/91│     150 │   7       │  │DATE     │ D │    8│  0 │
│ OUTBOX.DBF  │     673 │05/25/91│       2 │  11       │  │SENDER   │ C │   25│  0 │
│ INCOMING.DBF│   2,532 │05/25/91│      15 │  11       │  │RECEIVER │ C │   25│  0 │
│ OUTGOING.DBF│     619 │05/25/91│       4 │   7       │  │SUBJECT  │ C │   25│  0 │
│ USERS.DBF   │  46,391 │04/10/91│     771 │   3       │  │CONF     │ C │   27│  0 │
│ CONFS.DBF   │     256 │03/13/91│       9 │   1       │  │TEXT     │ M │   10│  0 │
│ FILES.DBF   │     259 │11/15/90│       0 │   7       │  │READ     │ L │    1│  0 │
└──────────────────────────────────────────────────────┘  └──────────────────────────┘
73,639 bytes in 7 files.                                   142 bytes per record,
                                                           11 fields.
```

Summary

Had enough yet? We've covered all the features of the TBrowse and TBColumn object classes, seen examples of how they are implemented, breezed through a number of handy TBrowse tool box functions, and saw how you can browse far more than just database fields, like arrays and text files. But even after all that we've barely scratched the surface! An entire book could be dedicated to the subject.

We sincerely hope this lengthy chapter inspires you not only to dive into TBrowse and create wonderful browse windows, but also to dig more deeply into the way Clipper's object-oriented design helps make the process so elegant. The Get System, which handles all the @..GET/READ-style data entry, and the Error System, which deals with run-time errors, are both implemented with this same object technology.

Reexamination 90/005,727

The Original

Page 1150 of Part V

Is Missing

Initial : AR
Date : 6-8-2000

CHAPTER 26

The GET Object Class

The basic point of every database management application is to store data in database files. But that data must first be entered (usually by a hapless data entry clerk who could not care less what language the software is written in). In the dBASE species (of which Clipper is a part), this data capturing mechanism is known as a **GET**. Briefly, programming data entry is a two-step process: The programmer displays one or more GETs upon the screen, each representing a data item that must be entered, and then issues a READ command, which activates the GETs and allows the user to enter various forms of data.

No development platform gives you more flexibility for GETs than Clipper 5. This unprecedented degree of control is due largely to Clipper 5's new GET object and the configurable GET system. In this chapter we will explore the inner workings of the GET object. We'll also demonstrate several creative ways to use the new WHEN clause, as well as add a MESSAGE to the @..GET command.

Overview

In prior versions of Clipper, issuing a READ statement was tantamount to handing over the reins to Clipper. We could occasionally resurface to execute VALID functions or hot key procedures, but for the most part we were stuck in the bowels of the READ command, completely at Clipper's mercy.

With the GET object and the reconfigurable GET system, the rules have changed. Clipper 5 lets us configure the GETs any way our heart desires. Unlike the "old way" (i.e., vanishing into READ limbo), we now maintain complete control at all times. With earlier versions of Clipper we had no access to the READ mechanism. However, not only does Clipper 5 give us access in the form of READMODAL() and supporting functions, but *it is written entirely in Clipper* (and thus very easy to modify if necessary). The source code for READMODAL() can be found in the file GETSYS.PRG, which is supplied with Clipper 5. (A discussion of the other functions in GETSYS.PRG can be found later in this chapter.)

The WHEN clause

Before we plunge into the brave new world of GET objects, let's talk about another less glamorous but tremendously useful Clipper 5 addition: the WHEN clause. In the same fashion as the VALID clause provides *post-validation* for each GET, the WHEN clause serves as *pre-validation*. If you specify a WHEN clause, it will be evaluated before you actually enter the GET, and if it returns false, it will prevent the GET from being edited.

In the example shown below, the user will not be able to enter the credit card number unless the **credit** variable is set true.

```
local fname := space(15), lname := space(15), credit := .f.
local cardno := space(20), custno := space(6)
@ 10, 20 say "First Name: " get fname
@ 11, 20 say "Last Name:  " get lname
@ 12, 20 say "Credit?     " get credit
@ 13, 20 say "Card Number:" get cardno when credit
@ 14, 20 say "Customer No:" get custno
read
```

The WHEN clause opens up a world of possibilities. Something as rudimentary as skipping a GET is really just the tip of the iceberg. The fun begins when you start calling functions from the WHEN clause. As with the VALID clause, all you need do is ensure that your WHEN clause evaluates to a logical value (which should be true (.T.) if you want the user to be able to enter that particular GET).

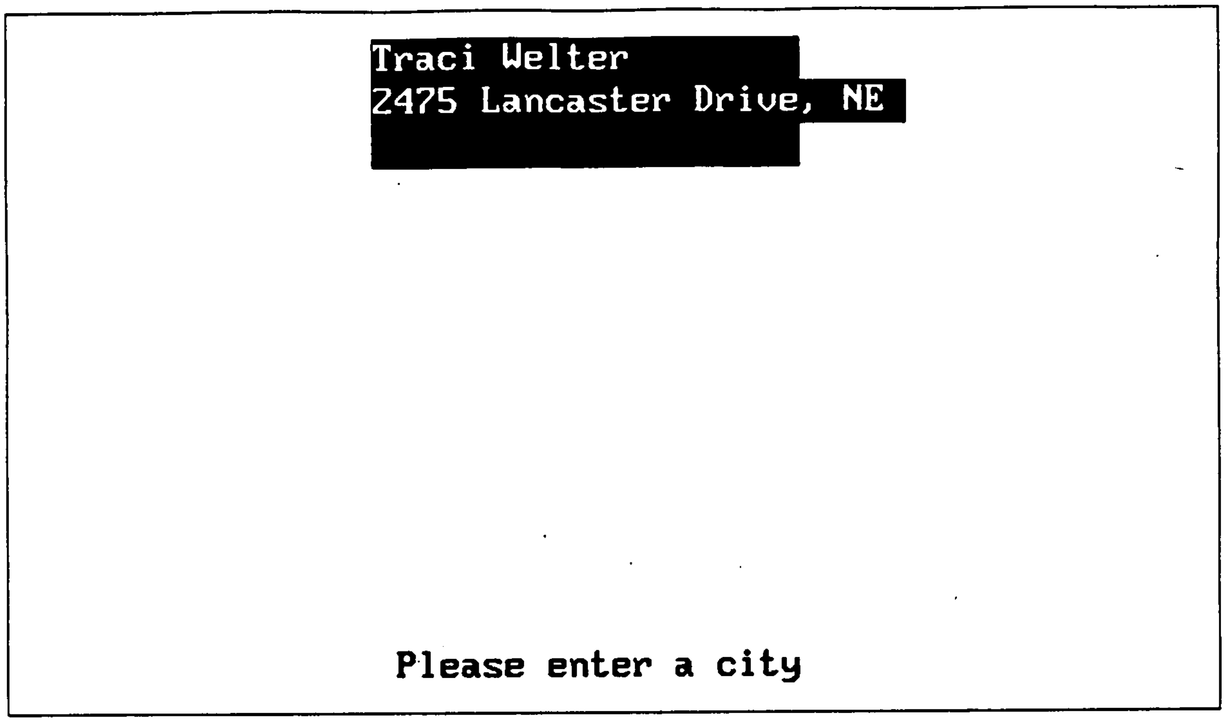
Listing 26.1 applies this simple principle to provide a message for each GET. Figure 26.1 shows this code in action.

Listing 26.1 Displaying messages with WHEN

```
function Main
memvar name, address, city
dbcreate('temp', { { 'name', 'C', 20, 0 } , ;
                   { 'address', 'C', 25, 0 } , ;
                   { 'city', 'C', 20, 0 } } )
use temp new
append blank
cls
name := temp->name
address := temp->address
city := temp->city
@ 10,0 get name    when FieldHelp(24, 1, "Please enter a name")
@ 11,0 get address when FieldHelp(24, 1,;
                                   "Please enter an address")
@ 12,0 get city    when FieldHelp(24, 1, "Please enter a city")
read
replace temp->name with name, temp->address with address,;
        temp->city with city
return nil
```

```
function FieldHelp(row, col, msg)
@ row, col say padr(msg, 50)
return .t.
```

Figure 26.1 GET using WHEN clause to display a message



```
Traci Welter
2475 Lancaster Drive, NE

Please enter a city
```

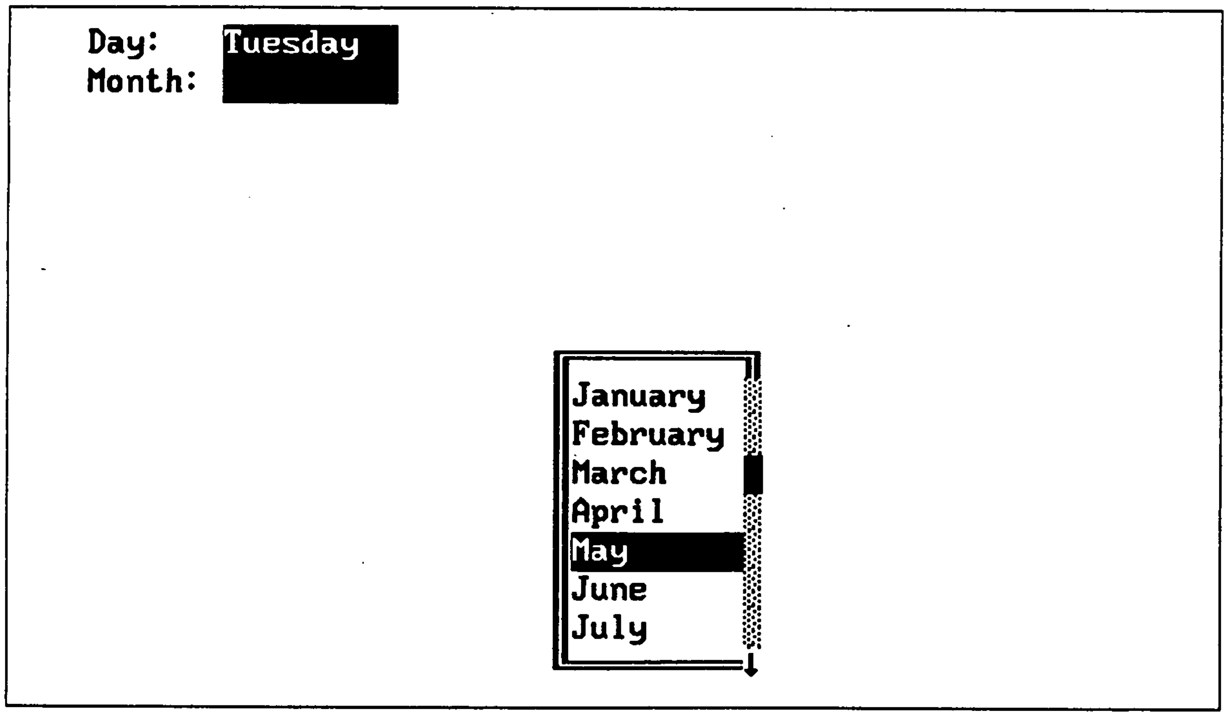
If you are using the Grumpfish Library, you probably already know about its APICK() function. APICK() makes selection from an array of available choices a breeze. Coupled with WHEN, APICK() becomes a devastating tool by which you let the user pick a value instead of typing it in. Listing 26.2 demonstrates this principle.

Listing 26.2 Displaying pop-up picklists with WHEN

```
function Main
local days := { "Monday", "Tuesday", "Wednesday", "Thursday",;
                "Friday"}
local mday := space(9), mmonth := space(9)
local months := {"January","February", "March", "April", "May",;
                "June","July","August", "September", "October",;
                "November", "December" }
cls
@ 0, 10 say "Day:  " get mday when ;
      ! empty(mday := days[ max(apick(9, 34, 15, 44, days), 1) ])
@ 1, 10 say "Month:" get mmonth when ;
      ! empty(mmonth := ;
              months[ max(apick(8, 34, 16, 44, months), 1) ] )
read
```

If you are not using Grumpfish Library, feel free to substitute ACHOICE() for APICK(). However, you may instead want to write your own user-defined function that calls ACHOICE() and handles all screen cleanup (as does APICK()). Figure 26.2 shows this code in action.

Figure 26.2 GET using WHEN to call a pop-up picklist



```
Day:    Tuesday
Month:

                    January
                    February
                    March
                    April
                    May
                    June
                    July
```

Note the careful construction of the WHEN clause. APICK() returns a numeric value that corresponds to the array element that was selected. If the user pressed the Esc key to leave APICK(), it will return 0. Because Clipper arrays are one-based rather than zero-based, a reference to array element 0 is a one-way ticket to DOS. Therefore, this possibility must be eliminated with the MAX() function, which in effect ensures that the minimum array element referred to will be one. We then assign the value of the appropriate array element to the variable and test it for emptiness. Since it will not be empty, the WHEN clause will return True, thus permitting the user to further edit the GET if required.

GETLIST

If you are compiling your Clipper 5 programs with the /**w** compiler option we discussed in Chapter 1, you have undoubtedly already made the acquaintance of the omnipresent GETLIST. This is a public array that Clipper 5 utilizes to maintain compatibility with earlier versions of Clipper. Whenever a @..GET command is issued, the preprocessor translates it into logic which creates a new GET object and adds it to the GETLIST array. When we issue the subsequent READ command, the contents of GETLIST are passed to the Clipper READMODAL() function, which then initiates the traditional full-screen edit. Thus, each GET object is an element in the GETLIST array, and the Clipper GET system moves between them as we press navigation keys (which you can completely redefine if you wish). When the READ process ends (i.e., when READMODAL() transfers control back to the calling program), the GETLIST array is cleaned out for next time (unless you specified the READ SAVE option).

Nested READs

Having access to the GETLIST array means that you will never again suffer a migraine trying to finesse nested READs. One easy way to nest READs is to declare a GETLIST array as **local** to each procedure that issues a READ command.

The source code in Listing 26.3 demonstrates this logic by nesting a READ in a VALID clause.

Listing 26.3 Example of nested READ

```
memvar getlist // to squelch compiler warnings

function Main
local x := 0, mdate := date() + 14
cls
@ 10, 10 say "Balance: " get x picture '#####.##' valid Credit(@x)
@ 11, 10 say "Due date:" get mdate
read
return nil

function Credit(balance)
local x := 0, getlist := {}, oldscrn := savescreen(10, 40, 10, 64)
@ 10, 40 say "Credit (if any):" get x picture '#####.##'
read
balance -= x        // subtract credit from original balance
restscreen(10, 40, 10, 64, oldscrn)
return .t.
```

You should already have a good grasp on the concept of file-wide statics (previously discussed in Chapter 6, “Variable Scoping”, and Chapter 12, “Program Design”).

The functions shown in Listing 26.4 make use of a file-wide static array (or “stack”) to save and restore GETs. These are also included on the source code diskette for this book.

Listing 26.4 Stack-based functions to save/restore GETs

```
/* GFGETS.PRG — stack-based functions to save/restore GETs */
static getstack_ := {}
```

```
memvar getlist

// GFSaveGets(): save current gets
function GFSaveGets()
aadd(getstack_, getlist)
getlist := {}   /* clear out current gets */
return len(getstack_)

// GFRestGets(): restore last-saved gets
function GFRestGets(ele)
/* use LIFO (last item in array) if no parameter was passed */
ele := if(ele == nil, len(getstack_), ele)
/* preclude empty array */
if len(getstack_) > 0
  /* pull GETs from last element in array */
  getlist := getstack_[ele]
  /* truncate length of array only if using LIFO, i.e.,
     no param passed)*/
  if ele == len(getstack_) .and. pcount() == 0
     asize(getstack_, len(getstack_) - 1)
  endif
endif
return nil
```

These functions are most useful if you want to construct multi-screen data entry scenarios. They allow you to push and pop active GETs at will.

Notice that the getstack array will only be truncated if we are using Last In First Out (LIFO). If you use random access with GFRestGets() by passing a parameter, the function will assume that you do not want to lop off the last set of GETs. (Imagine the chaos if you restored the first set of GETs, and the last set got trashed! Not a pretty picture.)

The code shown in Listing 26.5 demonstrates this by establishing three parallel sets of GETs, pushing each of those onto our GET stack with GFSaveGets(), and popping them off in random order with GFRestGets().

Listing 26.5 Parallel GETs with stack-based functions

```
memvar getlist

function SetsOGets
local x := { 1, 2, 3, 4, 5, 6, 7, 8, 9}, y, z
for z := 1 to 3
   for y := 1 to 3
      @ y * 2, 0 get x[(z - 1) * 3 + y]
   next
   GFSaveGets()
next
cls

GFRestGets(2)
ReGet()
read

GFRestGets(1)
ReGet()
read

GFRestGets(3)
ReGet()
read

aeval(x, { | a | qout(a) } )
return nil

//  ReGet(): redisplay all active GETs
static function ReGet
```

```
aeval(getlist, { | get | get:display() } )
return nil
```

If you are scratching your head about that ReGet() function, it is used to redisplay all active GETs. It makes use of the Display() method, which we will discuss in more detail later.

Creating GET objects

There are two ways to create Clipper 5 GET objects. The first and most common method is to let the preprocessor do the work for you in the form of the @..GET command. The second is to use the Clipper GETNEW() function.

@..GET

The best way to fully understand this critical process is to review the syntax of the @..GET user-defined command.

```
/* STD.CH */
#command @ <row>, <col> GET <var> [PICTURE <pic>];
                [VALID <valid>] [WHEN <when>] =>      ;
              SetPos( <row>, <col>) ; ;
              aadd(GetList, _GET_( <var>, <(var)>, <pic>, ;
              <{valid}>, <{when}>) )
```

Now let's write a short sample program that calls this command. We can then examine each clause of the @..GET command to see how the preprocessor acts upon it. If you are not already familiar with the various preprocessor result-markers and their actions, now would be a good time for you to brush up on them. (See Chapter 7, "The Preprocessor.")

Original (PRG)

```
@ 20,0 get x picture '###' valid ! empty(x) when y > 5
```

Preprocessed (PPO)

```
SetPos(20, 0) ; ;
aadd(GetList, _GET_(x, "x", "###", {|| ! empty(x)}, {|| y > 5}) )
```

First, the row and column positions are output using the normal result-marker into a call to the SetPos() function. This will position the cursor at row 20, column 0. This is crucial because the current cursor position will be used to determine the position of the GET object at the instant it is created. Therefore, the cursor must be moved to the desired position before the GET object is created.

The preprocessor then acts upon all of the other clauses in order to create a call to the internal Clipper function _GET_(). First, the name of the GET variable is output twice, once with the normal result-marker and again with the smart stringify result-marker:

```
_GET_( x, "x", ...)
```

The PICTURE clause is output with the normal result-marker since it is already in the form of a character string:

```
_GET_(x, "x", "###", ...)
```

The VALID (post-validation) clause is output with the blockify result-marker, because the GET system will expect this to be in the form of a code block so that it can evaluate it if necessary.

```
_GET_(x, "x", "###", {|| ! empty(x)}, ...)
```

Finally, the WHEN (pre-validation) clause is output with the blockify result-marker. Like the VALID clause, the GET system expects this to be in the form of a code block so that it can evaluate it if necessary.

```
_GET_(x, "x", "###", {|| ! empty(x)}, {|| y > 5})
```

The final step is to add this new GET object to the end of the GETLIST array, and thus increase GETLIST's length by one element.

Something that's not readily obvious here is that _GET_() also creates a code block that will be used to manipulate the variable. GET objects change the values of the variables by evaluating this code block, rather than manipulating the variables directly. (An example of the structure of such a code block can be found below under the discussion of the block instance variable.)

As with the current cursor position, the current color setting is taken into account when creating a GET object. The current unselected and enhanced color settings will be used when the GET object is unselected and selected, respectively. For example, if this is our current color setting:

```
setcolor("w/n, +w/r, , , w/b")
```

the GET object will be displayed in white on blue when unselected and high white on red when selected. You can also change the colorSpec instance variable tied to the GET object after it is created. With the 5.01 release of Clipper, you can use the optional COLOR clause to change colorSpec at the same time that the GET object is created (see below). We will discuss this in greater detail below.

Warning: _GET_() should never be called directly! All Clipper functions that begin with an underscore are for internal use only, and are always subject to change. If you need to create a GET object without using @..GET, use GETNEW().

New Clipper 5.01 @..GET clauses

As mentioned above, you can now specify a color for the GET using the optional COLOR clause. If you wish to use this clause, your color setting should be in the form of <standard>,<enhanced>. The <enhanced> color will be used when the GET is highlighted, and the <standard> color will be used when it is not. For example, if you wanted your GET to be white on red when highlighted, and white on blue when not highlighted, you could use syntax such as the following:

```
@ 20,0 get x color "w/b,w/r"
```

You can use the SEND clause to "send" methods to GET objects. For example, the STD.CH header file uses SEND as an intermediate step for implementing the @..GET..COLOR clause, as shown in Listing 26.6:

Listing 26.6 Extract from STD. CH @ ..GET..COLOR clause using SEND

First Step:

```
#command @ <row>, <col> GET <var>                ;
                        [<clauses,...>]          ;
                        COLOR <color>            ;
                        [<moreClauses,...>]    ; ;
      => @ <row>, <col> GET <var>                ;
                        [<clauses>]              ;
                        SEND colorDisp( <color>) ;
                        [<moreClauses>]
```

Second Step:

```
#command @ <row>, <col> GET <var>                ;
                        [PICTURE <pic>]          ;
                        [VALID <valid>]          ;
                        [WHEN <when>]            ;
                        [SEND <msg>]           ; ;
                     => SetPos( <row>, <col> ) ; ;
   aadd(GetList, ;
   _GET_( <var>, <(var)>, <pic>, <{valid}>, <{when}> ) ) ;
      [; atail(GetList):<msg>]
```

Do not feel compelled to use this if you do not fully understand its significance. Just know that it is there if/when you need it.

GETNEW()

GETNEW() creates a GET object without using the @..GET command. The syntax for GETNEW() is similar, but not identical, to what the preprocessor sends to the internal _GET_() function:

```
GetNew([<row>], [<column>], [<block>], [<var>],;
                      [<picture>], [<color>])
```

<row> and **<column>** are numeric expressions that represent the starting row and column position of the GET on the screen.

<block> is a "get..set" code block for the variable to be gotten. As mentioned above, the value of the variable will be manipulated via this code block.

<var> is a character expression representing the name of the GET variable. (At press time, this parameter is not in the Clipper documentation.)

<picture> is a character expression representing the PICTURE clause to use for the GET. If you do not pass this, it will be initialized to nil.

<color> is a character expression representing the color setting to use for the GET. If you do not pass this, the current unselected and enhanced color settings will be used as previously mentioned.

All of these parameters are optional. You may supply any or all of them to GETNEW(), or you may wait to assign them later.

The major differences between GETNEW() and _GET_() are:

GETNEW()	Lets you pass the color setting as a parameter, whereas _GET_() merely uses the current color setting.
GET()	Creates the code block for the GET variable automatically, and GETNEW() does not.

GET() accepts the code blocks for pre-validation (WHEN) and post-validation (VALID) as parameters. If you want to use these with a GET object created by GETNEW(), you must manipulate the instance variables after the fact.

However, once again we urge you never to call _GET_() directly! These differences are listed for illustrative purposes only.

Instance variables

Occasionally you will need to peek into the GET object. For example, is there a WHEN (pre-validation) or VALID (post-validation) clause tied to a specific GET? Do you want to change the PICTURE clause, COLOR setting, or even the row and column position of a GET? For the answers to these, and many other questions, you would refer to an Exported Instance Variable. Instance variables can be considered as global settings that each GET object lugs around. They may be polled as often as necessary to retrieve their values.

Table 26.1 lists all instance variables associated with GET objects. The values of instance variables marked by an asterisk ("*") can be re-assigned.

Table 26.1 GET Instance Variables

Name	*Purpose*	*Type*
badDate	Checks editing buffer for an invalid date	L
block*	Code block to associate GET with a variable	B
buffer*	Character string that contains the editing buffer	C
cargo*	User-definable variable (a real jumping-off point!)	?
changed	Checks if Get:buffer was changed	L
col*	GET column number	N
colorSpec*	Color for a GET	C
decPos	Decimal point position within the editing buffer	N
exitState*	Tracks the means by which a specified GET was exited	N

Name	*Purpose*	*Type*
hasFocus	Is this GET highlighted? (does it have "input focus"?)	L
name*	GET variable name	C
original	Character string containing original value of the GET	C
picture*	PICTURE string	C
pos*	Current cursor position within the editing buffer	N
postBlock*	Code block to validate a newly entered value (VALID)	B
preBlock*	Code block to decide if editing is permitted (WHEN)	B
reader*	Custom codeblock that determines how GET is edited	B
rejected	Was last insert/overStrike character rejected?	L
row*	GET row number	N
subscript	Returns subscript(s) if GET variable is an array element	A
type	GET variable data type	C
typeOut	Did user try to move cursor out of editing buffer?	L

C = character
N = numeric
L = logical
B = code block
A = array

To refer to any of these instance variables, you first specify the GET object in question, followed by a colon and the instance variable. We will use **g:** as the GET object identifier.

g:badDate

This instance variable contains a logical value. Its value will be false (.F.) unless you are GETting a date variable and have entered an invalid date in the editing buffer.

g:block (assignable)

This is the retrieval/assignment code block that the GET object uses as an intermediary to manipulate the value of the variable. It is created automatically by the internal Clipper _GET_() function, and you must create it yourself if you are using GETNEW(). The following example provides a good template for such a code block.

```
local x := space(20)
devpos(20, 10)
theget := getnew(row(),col(),;
                 { | val | if(pcount() > 0, x := val,x) })
```

Your code block must be constructed to accept an argument (in this case, **val**). It then must test whether or not the argument was passed using either PCOUNT() > 0 or **val** = NIL. If the argument is passed, its value should be assigned to the GET variable. If not, use the value of the GET variable.

Therefore, the code block can be EVALuated with or without arguments to either assign or retrieve, respectively, the value of the GET.

```
EVAL(block)     // retrieve value of GET variable
EVAL(block, 5)  // assign 5 to GET variable
```

g:buffer

As you edit a GET, you are making changes neither to the GET variable nor the code block. Rather, you are changing the editing buffer, which is a character string no matter what type of variable you are GETting. The type conversion is done internally at the time that the buffer contents are assigned to the GET variable with the Assign() method (which we will discuss below).

g:cargo (assignable)

This is an entirely user-defined instance variable. It may be used for various and sundry things, such as a message to be displayed whenever the GET is highlighted. (In fact, we will demonstrate this usage a bit later in this chapter.)

The code snippets shown in Listing 26.7 assign a message to **g:cargo**.

Listing 26.7 Storing messages in g:cargo

```
local x := date()
@ 20, 0 get x
getlist[1]:cargo := "Enter employee's birthdate"
read
```

As you become more bold and creative with the GET objects and GETSYS, you will discover that the best thing to put in **g:cargo** is an array, because it can then hold multiple values and thus be used for multiple purposes.

g:changed

This instance variable contains a logical value based on whether the GET editing buffer has been changed. Its value will be true if the buffer has been changed, or false if it has not.

g:col (assignable)

This contains a numeric that represents the column at which the GET will be displayed on the screen. When a GET object is created with @..GET the current screen column position is used to initialize this instance variable. You may change it after the fact, as the following code fragment demonstrates.

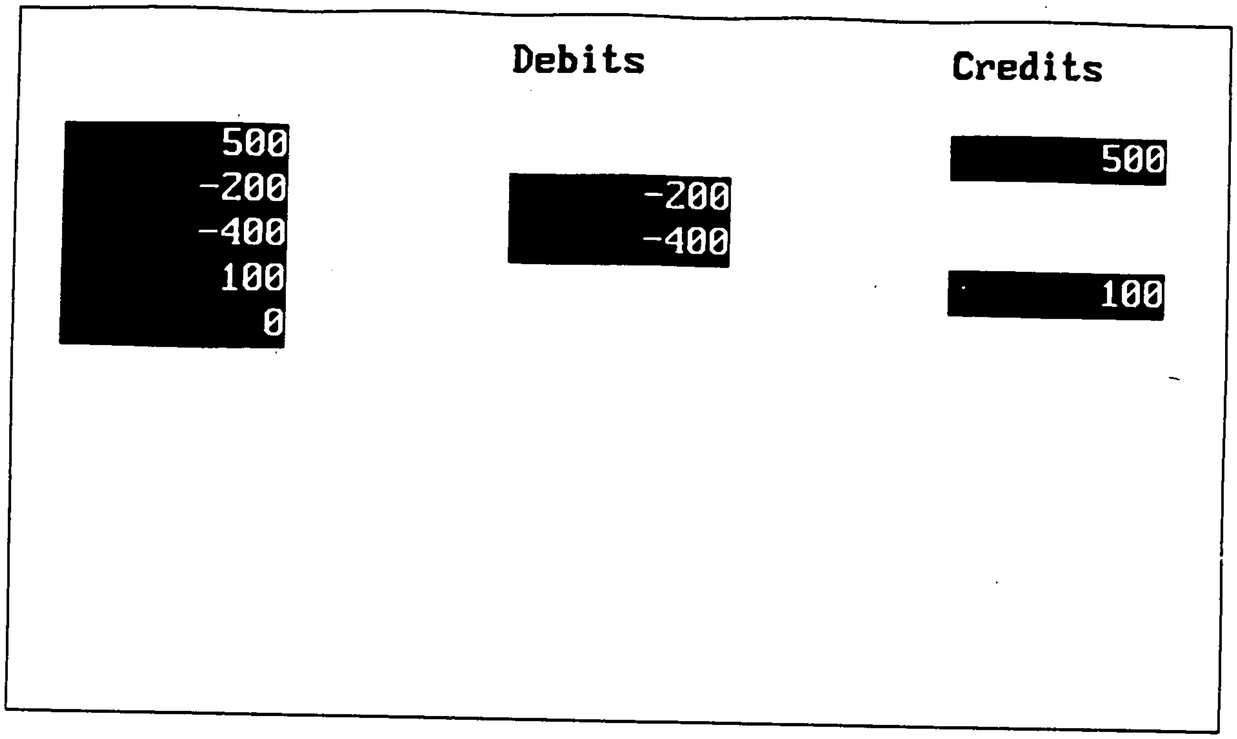
```
local x := 0
@ 20,10 get x          //  initially displays GET at @ 20,10
getlist[1]:col := 50 // GET will be redisplayed at @ 20,50
read
```

The code shown in Listing 26.8 uses the **g:col** instance variable to move each GET into the Debit or Credit column, dependent upon its value. Figure 26.3 shows sample output from this code.

Listing 26.8 Using g:col instance variable to move GETs

```
function Main
local a := { 0, 0, 0, 0, 0 }
cls
@ 8, 30 say "Debits"
@ 8, 50 say "Credits"
@ 10,10 get a[1] valid DebitCredit()
@ 11,10 get a[2] valid DebitCredit()
@ 12,10 get a[3] valid DebitCredit()
@ 13,10 get a[4] valid DebitCredit()
@ 14,10 get a[5] valid DebitCredit()
read
return nil

function DebitCredit
if getactive():varGet() > 0
   getactive():col := 50
else
   getactive():col := 30
endif
return .t.
```

Figure 26.3 Changing g:col instance variable



```
500
-200
-400
100
0
```

```
-200
-400
```

```
500

100
```

g:colorSpec (assignable)
This instance variable is a character string that dictates the color in which the GET will be displayed. The format for **g:colorSpec** is <unselected>,<selected>. When a GET object is created with @..GET, the current unselected and enhanced color settings are concatenated and used to initialize **g:colorSpec**. However, you may change it after the fact.

The sample code in Listing 26.9 illustrates how you could change the color of a GET within a VALID clause using the **g:colorSpec** instance variable. This principle could be used to call special attention to certain pieces of data.

Listing 26.9 Changing GET color in VALID clause

```
function ColorGet()
local x := 0
setcolor("w/n, w/r, , , w/b")
@ 20,10 get x valid Test()
? getlist[1]:colorSpec   // W/B,W/R
read
return nil

static function Test()
local ret_val := .t.
if getactive():varGet() == 0
   // make GET white on cyan when highlighted,
   // black on cyan when not
   getactive():colorDisp("N/BG, +W/BG")
   ret_val := .f.
endif
return ret_val
```

g:decPos

This instance variable contains a numeric value representing the position of the decimal point in the GET buffer. In the following code fragment **g:decPos** will have a value of 4.

```
local x := 0
@ 20,0 get x picture '###.##'
read
```

g:exitState (assignable)

This instance variable was added with the 5.01 release. It contains a numeric value that indicates how a GET was exited. It is used extensively within GETSYS.PRG.

Table 26.2 lists the possible **g:exitState** values, along with their manifest constants and the key(s) responsible for setting these values. Note that these are contained in the GETEXIT.CH header file. Always refer to the manifest constants rather than the numeric values, because the numerics are subject to change.

Table 26.2 g:exitState values

Value	*Manifest Constant contained in GETEXIT.CH*	*Responsible Key(s)*
0	GE_NOEXIT	Ha ha! There is no exit!!!
1	GE_UP	UpArrow
2	GE_DOWN	DnArrow
3	GE_TOP	Ctrl-Home
4	GE_BOTTOM	Ctrl-End
5	GE_ENTER	Enter
6	GE_WRITE	PgUp, PgDn, Ctrl-W
7	GE_ESCAPE	Esc
8	GE_WHEN	(WHEN clause tested False)

When you begin constructing your alternate GetReader() functions, remember to #include "getexit.ch" because you will need it.

g:hasFocus

This instance variable contains a logical value. It will be true when the GET object is highlighted (has input focus), and false if it's not. Input focus is assigned via the **g:setFocus()** method, which is called from within GetReader(). The following code fragment demonstrates how **g:hasFocus** changes value.

```
local x := 0
@ 20,10 get x
? getlist[1]:hasFocus // .F.
inkey(0)
getlist[1]:setFocus()
? getlist[1]:hasFocus // .T.
```

g:name (assignable)

This is a character string containing the name of the GET variable. It is initialized automatically when creating GET objects with @..GET, and must be passed as a parameter to GETNEW(). The name is used for identification purposes. It is assigned to READVAR() when the GET is given input focus.

Before you get any crazy ideas in that head of yours, you should know that changing the name of a GET variable will not affect which variable is manipulated. The previously-created code block attached to the GET object has the last word in this matter. The following fragment demonstrates this principle — although the **g:name** instance variable is changed to **y**, the variable that you manipulate will still be **x**. Press F1 while in the GET to test the current value of READVAR().

```
function Main
local x := 0, y := 5
set key 28 to TestReadVar
@ 20,10 get x
getlist[1]:name := 'y'
read
? x      // whatever you changed it to
? y      // still 5
return nil

function TestReadVar
@ 0,0 say "READVAR() := " + readvar()   // Y, not X
return nil
```

g:original

This instance variable is a character string that contains a copy of the original value of the GET buffer. If necessary, the contents of the GET buffer can be reset to their original contents with the Undo() method. This occurs in the GetApplyKey() function of GETSYS.PRG when you press Esc to exit a GET.

```
/* excerpted from GetApplyKey() */
   case (key == K_ESC )
   if (Set(_SET_ESCAPE) )
      get:undo()
      get:exitState := GE_ESCAPE
   endif
```

g:picture (assignable)
This is a character string that specifies which PICTURE clause, if any, to use for displaying the GET buffer. It will be assigned if you specify a PICTURE clause with @..GET or if you pass the appropriate parameter to GETNEW(). If not assigned, it will hold the value of NIL. You may always change it after the fact, as the following fragment demonstrates. Note that because the PICTURE clause is shortened from the default nine characters for numerics, the first GET must be erased to avoid screen sloppiness.

```
local x := 0
@ 20,10 get x
? getlist[1]:picture       // NIL
inkey(0)
scroll(20, 10, 20, 20, 0)  // clear old GET
getlist[1]:picture := '###.##'
read
```

g:pos (assignable)
This instance variable contains a numeric value representing the current cursor position in the editing buffer. The following example allows you to test this for yourself by pressing F1 while in the GET.

```
local x := "Press F1 to see where you are"
set key 28 to TestPos
@ 20,0 get x
read
return nil
```

```
function TestPos
@ 0,0 say "Current cursor position: " + str(getlist[1]:pos)
return nil
```

The example shown in Listing 26.10 is a bit more ambitious. It allows you to insert a name in the GET at the current cursor position. Move the cursor to the comma position, then press F2 to pop up the picklist of names. After you have selected a name from the picklist, we rely upon the **g:right()** method to move the cursor to the right (in exact correspondence with the length of the name that was inserted). This ensures that the cursor will be restored to its original position prior to the insert.

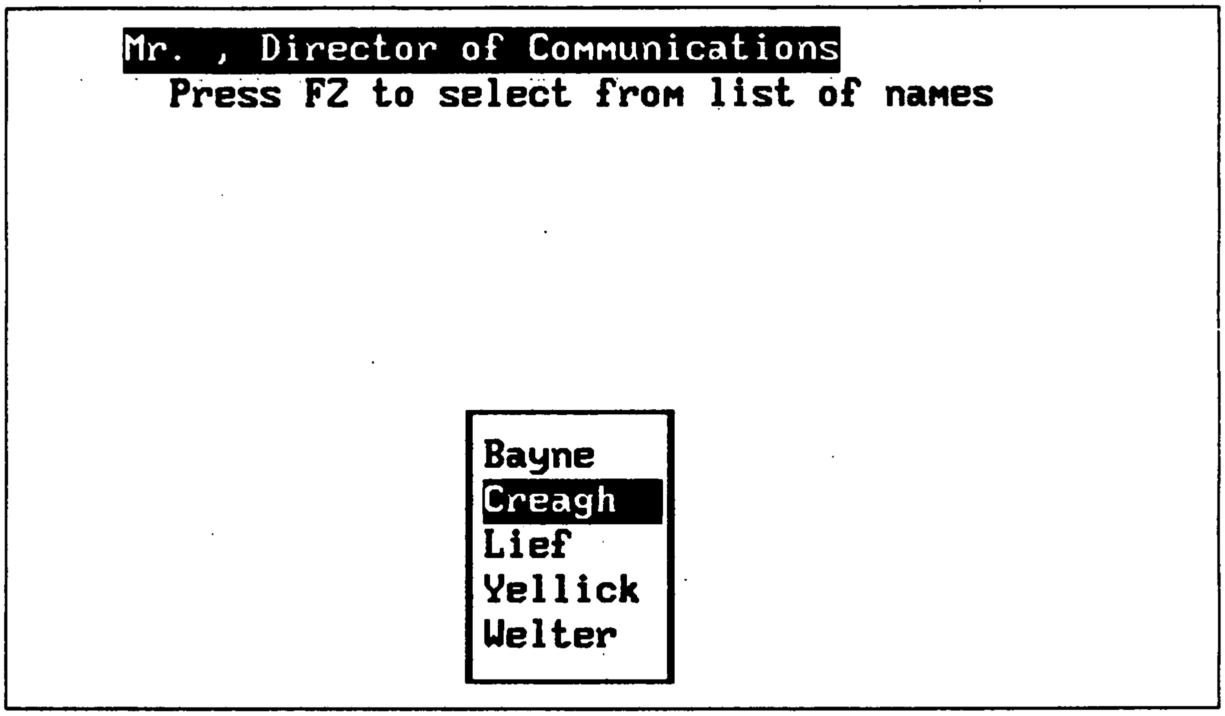
Listing 26.9 Inserting data into a GET

```
function Main
local m_var := "Mr. , Director of Communications"
set key -1 to ShowFields
cls
@ 1, 20 get m_var
@ 2, 22 say "Press F2 to select from list of names"
read
? m_var
return nil

static function ShowFields(a, b, c)
local position := getactive():pos, val := getactive():varGet()
local names := { "Bayne", "Creagh", "Lief", "Yellick", "Welter" }
local ele := 0, xx, oldscrn := savescreen(9, 35, 15, 44)
@ 9, 35 to 15, 44
/* don't allow Esc, which would cause array access error */
do while ele == 0
   ele := achoice(10, 36, 14, 43, names)
enddo
/* drop selected name into GET at current position */
getactive():varPut(substr(val, 1, position - 1) + names[ele] + ;
                   substr(val, position))
```

```
/* move cursor to original location in GET buffer, based on
   length of the name we just inserted */
for xx := 1 to len(names[ele] )
   getactive():right()
next
restscreen(9, 35, 15, 44, oldscrn)
return nil
```

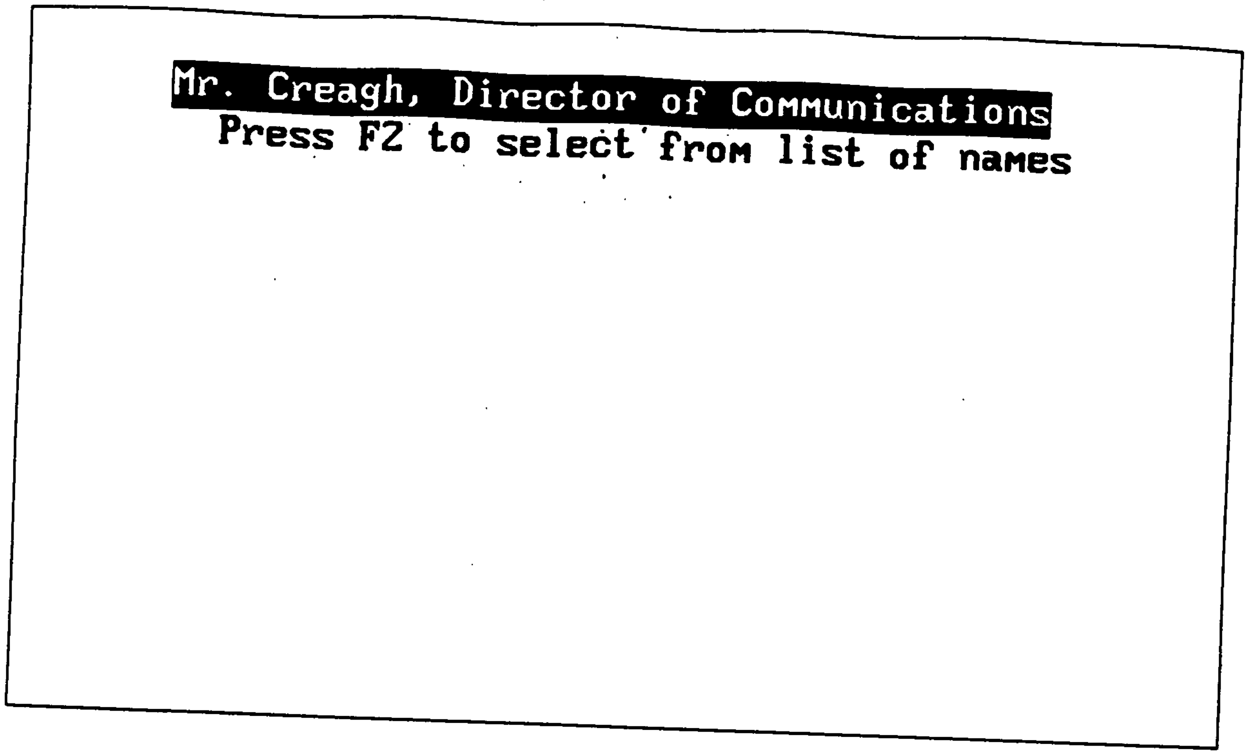
Figures 26.4 and 26.5 show the picklist of names, and the modified original GET, respectively.

Figure 26.4 Using g:pos instance variable to paste into a GET



```
Mr.  , Director of Communications
  Press F2 to select from list of names

Bayne
Creagh
Lief
Yellick
Welter
```

Figures 26.5 GET displaying selected name at g:pos


```
Mr. Creagh, Director of Communications
  Press F2 to select from list of names
```

g:postBlock (assignable)
This contains a code block used to "post-validate" the entry of a get. If you specify a valid clause with @..GET, it will be converted to a code block and stored in the **g:postBlock** instance variable. If you do not use a VALID clause, **g:postBlock** will contain the valid of a NIL. After exiting a GET, GetPostValidate() tests **g:postBlock** to see if a post-validation clause has been specified, and if so, evaluates the code block (passing the current GET object as a parameter). You may change **g:postBlock** after the fact if you wish.

In the following example we initialize **g:postBlock** with the VALID clause. It will initially evaluate to true because **x** is greater than 50. But **g:postBlock** is then changed so that **x** will not satisfy its condition.

```
local x := 100
@ 20, 0 get x valid x > 50
? eval(getlist[1]:postBlock)   // .T.
```

```
inkey (0)
*/ change to the equilvalent of VALID x > 200 */
get list[1]:postBlock := { | | x > 200 */
? eval(getlist[1]:postBlock)   // .F.
inkey(0)
read
```

g:preBlock (assignable)

This instance variable is similar to **g:postBlock**, except that it is evaluated before you enter a GET instead of after you exit. If you specify a WHEN clause with @..GET, it will be converted to a code block and stored in **g:preBlock**. If you do not use a WHEN clause, **g:preBlock** will contain the value of NIL. Before you enter a GET, GetPreValidate() tests **g:preBlock** to see if a pre-validation clause exists, and if so, evaluates the code block to determine whether the GET can be entered (passing the current GET object as a parameter). As with **g:postBlock**, you may alter **g:preBlock** at will.

In the following example, **g:preBlock** begins life as a NIL (didn't we all?). We then change it to a condition that tests false, and thus the GET is bypassed when we enter READMODAL().

```
local x := 100, y := .f.
@ 20, 0 get x
@ 21, 0 get y
? getlist[1]:preBlock   // NIL
inkey(0)
/* change to the equivalent of WHEN y */
getlist[1]:preBlock := { | | y }
? eval(getlist[1]:preBlock)   // .F.
inkey(0)
read
```

g:reader (assignable)

This instance variable was added with release 5.01. You can use it to implement special READ behaviors for any GET.

If **g:reader** contains a code block, READMODAL() will evaluate that block in order to read the GET (the GET object is passed as a parameter to the block). The block may in turn call any desired function to provide custom editing of the GET. If **g:reader** does not contain a code block, READMODAL() uses a default READ procedure for the GET.

g:Reader allows individual GETs to have specialized READ behaviors without requiring you to modify the standard READMODAL() function. This maintains compatibility for GETs which are to be handled in the customary fashion, as well as eliminating potential conflicts between different extensions to the GET/READ system.

Later in this chapter we will present an alternative GetReader() function that you can address via the **g:reader** instance variable. We will also discuss all of the supporting functions contained in GETSYS.PRG. These will assist you in creating your own alternate GET reading mechanisms.

g:rejected

This contains a logical value that indicates whether the last character specified by either the **g:insert()** or **g:overstrike()** method actually went into the editing buffer. It will contain false if the character was placed into the buffer, or true if it was rejected. Any subsequent text entry message resets this instance variable.

g:row (assignable)

The **g:row** instance variable contains a numeric that represents the row at which the GET will be displayed on the screen. When a GET object is created with @..GET, the current screen row position is used to initialize this instance variable. You may change it at any time, as shown in the following code fragment.

```
local x := 0
@ 20,10 get x         //  initially displays GET at @ 20,10
getlist[1]:row := 11  // GET will be redisplayed at @ 11,10
read
```

g:subscript

This instance variable (added with release 5.01) rectifies a long-standing problem with Clipper, namely the inability to identify which array element you were GETting. This caused great problems when you were attempting to tie context-specific help screens to each GET. If you are GETing an array element, **g:subscript** will contain an array of numeric values. This array will contain one element for each necessary subscript. (Please see the example below for more illumination.) If the GET is not an array element, **g:subscript** will contain NIL.

The following snippet demonstrates the three possible return values of the **g:subscript** instance variable.

```
local x := 0, y := { 1, 2, 3 }, z := { { 1, 2 }, { 3, 4} }
cls
@ 10, 10 get x
@ 11, 10 get y[3]
@ 12, 10 get z[1, 2]
? getlist[1]:subscript       // nil
? getlist[2]:subscript[1]    // 3
? getlist[3]:subscript[1]    // 1
? getlist[3]:subscript[2]    // 2
inkey(0)
read
```

The example below uses **g:subscript** to determine which GET you are on, and displays a help message accordingly. Although these help messages are hard-coded, this principle can easily be applied to a data-driven context-specific help system.

```
function Main
local a := {padr("John",20),padr("Doe",20),;
            padr("123 Main Street",30), ;
            padr("Anytown", 25), "MD", padr("21157", 10) }
```

```
set key 28 to HelpMe
cls
@ 10,0 get a[1]
@ 11,0 get a[2]
@ 12,0 get a[3]
@ 13,0 get a[4]
@ 14,0 get a[5]
read
return nil

static function HelpMe(p, l, v)
local helpmsg := {"first name","last name",;
                  "address","city","state","zip"}
if ! empty(getactive():subscript)
   @ maxrow(), 0 say "This is the " +
                     helpmsg[getactive():subscript] + ;
                     "... press a key"
   inkey(0)
   scroll(maxrow(), 0, maxrow(), maxcol(), 0)
endif
return nil
```

g:type

This instance variable is a character string denoting the type of the GET variable. This information can be useful to know if you want to act upon certain keypresses only for certain data types. For example, if you were to redefine the plus and minus keys to increment and decrement a GET variable, this would only be appropriate to dates and numerics. You could therefore test **g:type** before attempting to increment a character string (with predictably disastrous results).

The sample shown in Listing 26.11 establishes four GET objects and allows you to test each one for **g:type** by pressing F1. The TestType() function calls the Clipper GETACTIVE() function, which returns the currently active GET object. (See below for an interesting alternative.)

Listing 26.11 Testing GET types

```
function Main
local w := "Press F1 to test type of each GET",x :=0, y:=.t.,
z:=date()
set key 28 to TestType
cls
@ 11, 0 get w
@ 12, 0 get x
@ 13, 0 get y
@ 14, 0 get z
read
return nil

function TestType(p, l, v)
local type := { "Character", "Numeric", "Date", "Logical" } ;
                    [at(getactive():type, "CNDL")]
@ 0,0 say "Current variable is " + padr(type, 9)
return nil
```

If someone put a gun to your head and forced you to recode this function without using GETACTIVE(), you could actually survive! Do you remember the **g:name** instance variable? Because it is always part of the GET object, it gets assigned to the READVAR() function. The return value of READVAR() is then always passed as the third parameter to hot-key procedures. Therefore, this parameter can be used to perform an ASCAN() against the GETLIST array. When a match is found for the **g:name** instance variable, then we have found the current GET object.

```
function TestType(p, l, v)
local ele := ascan(getlist, { | get | upper(get:name) == v } )
local type := { "Character", "Numeric", "Date", "Logical" } ;
                    [at(getlist[ele]:type, "CNDL")]
```

This is further proof that, no matter what the problem, Clipper 5 gives you several different ways to solve it.

g:typeout

This instance variable contains a logical value. It will contain true if you attempt to move the cursor out of the editing buffer or if there are no editable positions in the editing buffer. Its value will be reset by any of the methods that move the cursor in the GET buffer.

The source code in Listing 26.12 thoroughly demonstrates manipulation of instance variables. It initializes a GET object with the @..GET statement, which can be referred to directly because we know its position in the GETLIST array (namely, numero uno). The instance variables are then manipulated to change the screen position, color, PICTURE, and VALID clause.

Listing 26.12 Manipulating instance variables

```
function Main
memvar getlist
local x := "this is a test"
cls
@ 10,30 get x picture '@!'
devpos(16, 0)              // move cursor to row 16, column 0
qout("GET row:   ", getlist[1]:row)              // output: 10
qout("GET column:", getlist[1]:col)              // output: 30
qout("GET color:", getlist[1]:colorSpec)         // output: N/W,N/W
qout("VALID clause:  ", getlist[1]:postBlock)    // output: nil
qout("PICTURE clause:", getlist[1]:picture)      // output: @!
qout()
qout('***Press any key to continue this example***')
inkey(0)
getlist[1]:row -= 5   /* change row position to 5 */
getlist[1]:col -= 20  /* change column position to 10 */
/* make GET white on blue when highlighted, white on red when not
*/
getlist[1]:colorSpec := "+W/R, +W/B"
```

```
/* add to the PICTURE clause */
getlist[1]:picture += "@K"
/* attach a VALID clause on-the-fly to prevent an empty field */
getlist[1]:postBlock := { | val | ! empty(val) }
read
```

Now you should have a better understanding of instance variables, so let's review three different ways to create the same GET object (shown in Listing 26.13).

Listing 26.13 Creating GET objects

```
/* this is common to all three examples */
local x := space(20), theget

// first example: GETNEW with all applicable parameters
theget := getnew(20, 10, { | val | if(pcount() > 0, ;
                x := val, x) }, "x", "@!", "N/BG, +W/BG")
readmodal( {theget} )   // must be passed as an array!

// second example: GETNEW with no parameters - this also
// reinforces the relationship between the parameters
// shown above and the instance variables.
theget := getnew()
theget:row := 20
theget:col := 10
theget:block := { | val | if(pcount() > 0, x := val, x) }
theget:name := "x"
theget:picture := "@!"
theget:colorSpec := "N/BG, +W/BG"
readmodal({theget} )  // again, must be passed as an array!

// third example: @..GET
@ 20, 10 get x picture '@!' color "N/BG, +W/BG"
read
```

After looking at the code involved, you might wonder why you would ever want to use GETNEW() over @..GET. There are two good situations. First, if you want absolute control over when the GET is actually displayed, you must use GETNEW() instead of @..GET. The second situation would be in conjunction with TBrowse.

Listing 26.14 demonstrates code for a generic database browser. A multi-dimensional array is created to hold customary keypresses and corresponding code blocks. Each of the keypresses triggers the expected action, except for Enter which is reserved for editing fields via the function GrabThatVar().

Listing 26.14 GETNEW() and TBrowse

```
#include "inkey.ch"

function main
local x, browse := TBrowseDB(3, 0, 15, 79), column, key, ele
local browseaction := { { K_LEFT, { | | browse:left() } } , ;
                     { K_RIGHT, { | | browse:right() } } , ;
                     { K_UP, { | | browse:up() } } , ;
                     { K_DOWN, { | | browse:down() } } , ;
                     { K_HOME, { | | browse:home() } } , ;
                     { K_END, { | | browse:end() } } , ;
                     { K_PGUP, { | | browse:pageUp() } } , ;
                     { K_PGDN, { | | browse:pageDown() } } , ;
                     { K_CTRL_LEFT, { | | browse:panLeft() } } , ;
                     { K_CTRL_RIGHT, { | | browse:panRight() } }, ;
                     { K_CTRL_HOME, { | | browse:panHome() } } , ;
                     { K_CTRL_END, { | | browse:panEnd() } } , ;
                     { K_CTRL_PGUP, { | | browse:goTop() } } , ;
                     { K_CTRL_PGDN, { | | browse:goBottom() } } }
setcursor(0)
use articles new
for x := 1 to fcount()
   column := TBColumnNew(field(x),;
                         fieldwblock(field(x), select()))
   browse:AddColumn(column)
next
```

```
do while key != K_ESC
   dispbegin()
   do while ! browse:stabilize() .and. (key := inkey() ) == 0
   enddo
   dispend()
   if browse:stable
      key := inkey(0)
   endif
   if key == K_ENTER and browse:stable
   GrabThatVar(browse)
   elseif (ele := ascan(browseaction, { | a | key == a[1] }) ) > 0
      eval(browseaction[ele, 2])
   endif
enddo
return nil

function GrabThatVar(b)
// determine current column object out of the browse object
local column := b:getColumn(b:colPos), mget,;
      oldcurs := setcursor(2)
// create a corresponding GET
mget := getnew(row(), col(), column:block, column:heading, ;
               b:colorSpec)
readmodal({mget} )  // must pass as an array!
setcursor(oldcurs)   // restore previous cursor
b:refreshCurrent()
return nil
```

As mentioned earlier, GETNEW() requires that you create the code block corresponding to the GET variable. GrabThatVar() does this by first working backwards from the TBrowse object to determine the current column. This is easy by polling the TBrowse **b:colPos** instance variable, and then passing that result to the TBrowse

b:getColumn method. This leaves us with a TBrowse column object, which as you should remember from Chapter 25, has a code block attached to it telling it what to display. This code block will serve as the basis for our new GET object.

IMPORTANT NOTE: If you want to give your user the ability to edit cells on-the-fly in this manner, be sure to load your TBrowse columns with retrieval/assignment code blocks. The Clipper FIELDBLOCK() and FIELDWBLOCK() functions are your best choice. If you have a simple code block such as "{| |customer->fname}" you will NOT be able to properly create a GET object from it!

Once the GET object is in hand, it is passed (as an element in a literal array) to READMODAL(). Are you wondering why we did not simply issue a READ command? Have a look and the answer should be obvious.

```
/* from STD.CH */
#command READ    => ;
                  ReadModal(GetList) ; GetList := {}
```

READ is predefined to pass GETLIST as a parameter. Because GETLIST is not being used in this situation, this would do no good at all. This also demonstrates that you do not always have to pass GETLIST to READMODAL(). You can pass any array containing one or more GET objects, which is why **mget** had to be passed as an element in an array. If **mget** was passed by itself (as opposed to being an element in an array), READMODAL() would pout and sulk.

Methods

GET objects are created and then passed to READMODAL(), which processes them using various *methods*. (For simplicity you can consider methods to be functions that are tied to GET objects.) These methods handle everything from moving the cursor within a GET to highlighting and unhighlighting GETs. (By the way, the acts of highlighting and unhighlighting a GET are known in Clipper 5-speak as "setting focus" and "killing focus," respectively.)

Table 26.3 lists methods that change the status of the GET object.

Table 26.3: GET status methods

Name	*Purpose*
assign()	Assigns editing buffer contents to GET variable
colorDisp()	Changes GET color and redisplays GET on screen
display()	Displays GETon the screen
killFocus()	De-highlights GET object
reset()	Resets internal state information of the GET
setFocus()	Highlights (gives input focus to) GET object
undo()	Sets the GET variable back to **g:original**
updateBuffer()	Updates editing buffer and redisplays the GET
varGet()	Returns the current value of the GET variable
varPut()	Sets the GET variable to the passed value

Table 26.4 lists methods that move the cursor within the get:buffer.

Table 26.4 GET cursor methods

Name	*Purpose*
end()	Move cursor to rightmost position in GET
home()	Move cursor to leftmost position in GET
left()	Move cursor left one character

right()	Move cursor right one character
toDecPos()	Move cursor to immediate right of decimal position (**g:decPos**)
wordLeft()	Move cursor left one word
wordRight()	Move cursor right one word

Table 26.5 lists methods used for editing the **g:buffer**.

Table 26.5 GET editing methods

Name	*Purpose*
backspace()	Moves cursor left and deletes one character
delLeft()	Deletes character to the left of the cursor
delRight()	Deletes character to the right of the cursor
delWordLeft()	Deletes word to the left of the cursor
delWordRight()	Deletes word to the right of the cursor
insert()	Inserts characters into the editing buffer
overStrike()	Overwrites characters in the editing buffer

To refer to any of these methods, you specify the GET object in question, followed by a colon and the method. For example, the following code would highlight the get object, i.e. "g", and move the cursor to the rightmost position within it:

```
g:setfocus()
g:end()
```

The best way to learn about each of these methods is to carefully scrutinize the source code in GETSYS.PRG.

g:colorDisp() note: This method was added with the 5.01 release of Clipper. It changes a GET object's colors and redisplays it. It is functionally equivalent to assigning the **g:colorSpec** instance variable and issuing the **g:display()** method. **g:colorDisp()** means that you can specify special colors for your GETs and have them take effect immediately (instead of only after the GET has been activated).

The optional @..GET..COLOR clause gets translated by the preprocessor into a call to **g:colorDisp()**, as shown below.

Original (.PRG):

```
@ 21,0 get y color "n/bg,+w/bg"
```

Preprocessed Output (PPO):

```
SetPos(21, 0) ; ;
aadd(GetList, _GET_(y, "y",,, ) ); ;
atail(GetList):colorDisp("n/bg,+w/bg")
```

Enhancements

During the preprocessor discussion in Chapter 7, much time was spent on the esoteric match-markers and result-markers, which are confusing. However, situations like the GET system will make that tribulation worthwhile. A thorough knowledge of the preprocessor's nuances are vital to performing the following feats of magic.

As already discussed, the @..GET command is preprocessed into several function calls that create the GET object and add it to the GETLIST array. This unprecedented degree of open architecture means that we can further benefit from the preprocessor to add enhancements to the @..GET command.

Structure of GETSYS.PRG

The 5.01 release of Clipper introduced a completely overhauled GETSYS.PRG. A number of the core functions used in the standard GET handler have been made **public** so that they can be used by your customized GET readers (which you could then implement via the **g:reader** instance variable described above). These functions are as follows:

GetReader(<oGet>)

GetReader() implements the standard read behavior for GETs. By default, ReadModal() uses the GetReader() function to read GET objects. GetReader() then calls other functions in GETSYS.PRG to do the work of reading the GET. Those functions are described below.

(Note that you can supersede GetReader() by assigning a code block to the **g:reader** instance variable, which was discussed above. This code block can be configured to call your own replacement GetReader()-type function. We will show an example of this below when we address the MESSAGE clause.)

GetApplyKey(<oGet>, <nKey>)

GetApplyKey() applies an INKEY() value to a GET object. Cursor movement keys change the cursor position within the GET, data keys are entered into the GET, and so on. If you want to implement special key handling (e.g., password or calculator-style data entry), you can write your own version of this function.

GetApplyKey() properly handles keystrokes that have "hot-key" procedures attached to them.

The GET object must be highlighted ("focused") before keys are applied.

Note: If CLEAR GETS is executed by a SET KEY, **g:exitState** will be set to GE_ESCAPE. In the standard system, this cancels the current GET without assigning the edited value and terminates ReadModal().

GetPreValidate(<oGet>)
GetPreValidate() validates the GET object for editing, including evaluating **g:preBlock** (WHEN clause) if present.

GetPreValidate() returns a logical return value: true (.T.) if the GET has been successfully pre-validated, or false (.F.) if not. The **g:exitState** instance variable is also set to reflect the outcome of the prevalidation:

Setting	*Description*
GE_NOEXIT	indicates pre-validation success, ok to edit
GE_WHEN	indicates pre-validation failure
GE_ESCAPE	indicates that a CLEAR GETS was issued

In the default GET system, a **g:exitState** of GE_ESCAPE cancels the current GET and terminates ReadModal().

GetPostValidate(<oGet>)
GetPostValidate() validates a GET after editing, including evaluating **g:postBlock** (VALID clause) if present. It returns a logical value indicating whether or not the GET has been successfully post-validated. If a CLEAR GETS is issued during post-validation, **g:exitState** is set to GE_ESCAPE and GetPostValidate() returns true (.t.).

GetDoSetKey(<oGet>)

This function executes a SET KEY block, preserving the context of the passed GET object. The procedure name and line number passed to the SET KEY block are based on the most recent call to ReadModal().

@..GET..MESSAGE

You have already seen how to tie messages to each GET with the WHEN clause and the **g:cargo** instance variable. The latter method is more succinct and more in line with the intended purpose of the WHEN clause (namely, pre-validation). However, when we showed this earlier, we had to manipulate the **g:cargo** instance variable directly, which was hardly an elegant solution. The preprocessor will make this much cleaner.

```
#xcommand @ <row>, <col> GET <var> [PICTURE <pic>] ;
         [VALID <valid>] [WHEN <when>] ;
         [MESSAGE <message>] => ;
            SetPos( <row>, <col>) ; ;
            aadd(GetList, _GET_( <var>, <(var)>, <pic>, ;
                 <{valid}>, <{when}>) ) ; ;
            Atail(getlist):reader := { | g | MyReader(g) } ;
            [; Atail(getlist):cargo := <message> ]
```

We have added two additional statements to this user-defined command. The first assigns the **g:reader** instance variable to call an alternate GetReader() function. This is necessary because GetReader() is where you would want to insert the code to display the messages. However, rather than hack up Nantucket's GetReader(), you should write your own version. That's where the reader instance variable comes into play.

The ATAIL() function (new with 5.01) refers to the last element in an array. Because this GET was just added to the GETLIST array, ATAIL() will be pointing to the new GET.

The second statement, which gets preprocessed only if you actually specify the MESSAGE clause, actually assigns the message to the **g:cargo** instance variable.

Listing 26.15 demonstrates the @..MESSAGE clause in action. It includes MyReader(), which serves as an alternative to the standard GetReader() function.

Listing 26.15 @..MESSAGE with g:cargo/g:reader instance variables

```
#xcommand @ <row>, <col> GET <var> [PICTURE <pic>] ;
          [VALID <valid>] [WHEN <when>] ;
          [MESSAGE <message>] => ;
             SetPos( <row>, <col>) ; ;
             aadd(GetList, _GET_( <var>, <(var)>, <pic>, ;
                   <{valid}>, <{when}>) ) ; ;
             atail(getlist):reader := { | g | MyReader(g) } ;
             [; atail(getlist):cargo := <message> ]

#include "getexit.ch"

function Test
local x := 0, y := 1, getlist := {}
set message to maxrow()
cls
@ 20,0 get x message "This is the first GET"
@ 21,0 get y
read
return nil

/*
     MyReader()
     Alternate modal read of a single GET.
*/
function MyReader(get)
local mess_row
// read the GET if the WHEN condition is satisfied
if (GetPreValidate(get))
   // activate the GET for reading
   mess_row := set(_SET_MESSAGE)
   if ! empty(get:cargo)
      @ mess_row, 0 say padc(get:cargo, maxcol() + 1)
   else
      scroll(mess_row, 0, mess_row, maxcol(), 0)
   endif
   get:SetFocus()
   do while (get:exitState == GE_NOEXIT)
      // check for initial typeout (no editable positions)
      if (get:typeOut)
         get:exitState := GE_ENTER
      endif

      // apply keystrokes until exit
      do while (get:exitState == GE_NOEXIT)
         GetApplyKey(get, Inkey(0))
      enddo

      // disallow exit if VALID condition is not satisfied
      if (!GetPostValidate(get))
         get:exitState := GE_NOEXIT
      endif
   enddo

   // de-activate the GET
   get:KillFocus()

endif
return nil
```

Our alternate GetReader() function calls many of the core functions in Nantucket's GETSYS.PRG. However, it would be a relatively simple matter for you to write your own replacements for these functions. For example, the following is an alternative GetApplyKey() function that enables step entry for dates and numerics. The user can simply press "+" to increment the value, or "-" to decrement. Note that this function respects any VALID clause to limit the range of values.

Listing 26.16 Alternate GetApplyKey() function for step entry

```
/*
         Function: GKeyStep()
         Purpose: Alternate GetApplyKey() function that allows
                  step entry for dates and numerics
         Excerpted from Grumpfish Library
         Author: Greg Lief
         Copyright (c) 1991 Greg Lief
         Dialect: Clipper 5.01
*/

#include "inkey.ch"
#include "getexit.ch"
procedure GKeyStep(get, key)
local cKey
local oldvalue
local bKeyBlock

// check for SET KEY first
if bKeyBlock != NIL
   GetDoSetKey(bKeyBlock, get)
else
   do case

      case ( key == K_UP )
         get:exitState := GE_UP

      case ( key == K_SH_TAB )
         get:exitState := GE_UP

      case ( key == K_DOWN )
         get:exitState := GE_DOWN

      case ( key == K_TAB )
         get:exitState := GE_DOWN

      case ( key == K_ENTER )
         get:exitState := GE_ENTER
      case ( key == K_ESC )
         if ( Set(_SET_ESCAPE) )
            get:undo()
            get:exitState := GE_ESCAPE
         endif

      case ( key == K_PGUP )
         get:exitState := GE_WRITE

      case ( key == K_PGDN )
         get:exitState := GE_WRITE

      case ( key == K_CTRL_HOME )
         get:exitState := GE_TOP

// both ^W and ^End terminate the READ (the default)
      case (key == K_CTRL_W)
         get:exitState := GE_WRITE

      case (key == K_INS)
         Set( _SET_INSERT, ! Set(_SET_INSERT) )
         setcursor( if(set(_SET_INSERT), 3, 1) )

      case (key == K_CTRL_U)
         get:Undo()

      case (key == K_HOME)
         get:Home()

      case (key == K_END)
         get:End()

      case (key == K_RIGHT)
         get:Right()

      case (key == K_LEFT)
         get:Left()

      case (key == K_CTRL_RIGHT)
         get:WordRight()
```

```
case (key == K_CTRL_LEFT)
   get:WordLeft()

case (key == K_BS)
   get:BackSpace()

case (key == K_DEL)
   get:Delete()

case (key == K_CTRL_T)
   get:DelWordRight()

case (key == K_CTRL_Y)
   get:DelEnd()

case (key == K_CTRL_BS)
   get:DelWordLeft()

otherwise

   if (key >= 32 .and. key <= 255)
      cKey := chr(key)
      // test for step entry on numerics and dates
      if cKey $ '-+' .and. get:type $ "ND"
         oldvalue := get:varGet()
      if cKey == "-"
         get:varPut(get:varGet() - 1)
      else
         get:varPut(get:varGet() + 1)
      endif
      // make sure we are still within valid range!
      if get:postBlock != NIL .and. ;
              ! eval(get:postBlock, get)
         get:varPut(oldvalue)
      endif
      get:updateBuffer()
   else
      if (get:type == "N" .and. ;
              (cKey == "." .or. cKey == ","))
```

```
                    get:ToDecPos()
                    else
                    if ( Set(_SET_INSERT) )
                       get:Insert(cKey)
                    else
                       get:Overstrike(cKey)
                    endif
                    if (get:typeOut .and. !Set(_SET_CONFIRM) )
                       if ( Set(_SET_BELL) )
                          ?? Chr(7)
                       endif
                       get:exitState := GE_ENTER
                    endif
                 endif
              endif
           endif
        endcase
     endif
     return
```

Summary

We covered a lot of uncharted territory in this chapter. You are now comfortable with GET objects and the GETLIST array. You can now use the WHEN clause (and not just for skipping GETs). You should understand how to create GET objects with the GETNEW() function. You should be able to query and change instance variables with the greatest of ease. Calling methods to change the status of the cursor or GET should be child's play to you.

Finally, if you think of something that you want your GETs to do but Clipper doesn't provide, you should now have both the courage and chutzpah to do it yourself! The sky is indeed the limit with the Clipper 5 GET system.

Reexamination 90/005,727

The Original

Page 1200 of Part V

Is Missing

Initial : AR
Date : 6-8-2000

CHAPTER 27

The Error Object Class

Clipper 5 provides programmers with precise control over run-time error handling and a way to construct otherwise impractical program logic. This chapter covers the use of the Error class of objects and other components used to control the run-time environment.

We'll start by laying the groundwork with BEGIN SEQUENCE/BREAK and the use of assertions. Then we'll move into the actual error object class and programming techniques. After reading this chapter you'll be able to construct programs that can recover efficiently from errors. You will be able to control completely what happens when errors occur.

Before jumping into error objects, see "Introduction to the Object Classes" that begins Chapter 25 for an overview of object oriented programming concepts.

Summer '87 review

In Clipper Summer '87, error handling is directed by a small collection of functions in ERRORSYS.PRG. You change the way Clipper handles run-time errors by altering these functions. When a run-time error occurs Clipper S'87 categorizes it and makes a call to one of the ERRORSYS functions along with a small number of parameters relevant to the error. There can be only one ERRORSYS.OBJ per application and consequently one version of each function. You must handle special cases by adding code to the ERRORSYS functions or by branching out of them into functions specific to your application.

```
Expr_Error(name, line, info, model, _1, _2, _3)
Undef_Error(name, line, info, model, _1)
Misc_Error(name, line, info, model)
Open_Error(name, line, info, model, _1)
Db_Error(name, line, info)
Print_Error(name, line)
```

Clipper 5 greatly improves on this concept of run-time error handling. Errors are now considered to be an instance of the error object class and can be manipulated in much the same way as other objects such as Get and TBrowse.

Starting simple with BEGIN SEQUENCE..END

The BEGIN SEQUENCE block structure is a two-edged sword that can cause a great deal of trouble. With it you can both solve and create a wide variety of problems. BEGIN SEQUENCE..END is used to delimit a block of source code that might cause trouble at run-time. The BREAK command will exit the current BEGIN SEQUENCE block (they can be nested) from anywhere in the application. Structured programming purists are shaking their heads and muttering, "that's almost as bad as a GOTO," and their paranoia should be taken into consideration. You can make a terrible mess of your program logic by misusing BEGIN SEQUENCE and BREAK. But if you promise to be careful we'll describe some situations where it solves problems in an elegant and efficient way.

To get the flavor of the programming technique, here is a common problem. Suppose you have a complex block of logical code, and deep into it you discover a situation which requires the whole thing to be shut down and exited. If you had known of the problem from the start you could have coded the logic differently, but now you're stuck. The code works perfectly except for the "exit all the way out" problem. Listing 27.1 gives a solution:

Listing 27.1 A very simple form of assertion function

```
major_error := .f.
begin sequence
  do while .t.
    if this .or. that
      for i := j to k
        do while foo > bar
          result := SomeCalc(i,j,k)
          if result = "ERROR"
            major_error= .t.
            break
          else
            DoSomeMoreWith(result)
          endif
        enddo
      next i
    else
      OtherStuff()
    endif
  enddo
end sequence
if major_error
  CleanUp("And Complain to User")
endif
```

When the BREAK is encountered the code branches execution to the next statement after the END SEQUENCE. In the above example it allowed us to cleanly exit all of the loops without complex signalling using loop control variables, like so:

```
while (foo > bar) .and. no_errors
```

Note the attempt to redeem ourselves by flagging the BREAK condition so it can be more easily tracked and debugged. The references to **major_error** are not at all necessary to use BREAK but from past experience debugging other people's code we assure you it's well worth the trouble.

So far we have not violated too many programming rules, mostly because we flagged the BREAK condition and kept the effects limited and local. Unlike other block-type commands (like IF..ENDIF and DO..ENDDO), the BREAK command can be issued against a BEGIN SEQUENCE..END block outside of the procedure or function that contains it. This can definitely lead to a debugging nightmare, but it can also be put to good use for controlling otherwise uncontrollable run-time errors.

Breaking a Bad Habit with Compiler Error C2086
Both Clipper Summer '87 and the initial release of Clipper 5 allowed a serious structural error to slip past the compiler and into your final application: a RETURN from within a BEGIN SEQUENCE..END block. By allowing the application to jump out of the block, the block is left unterminated, setting the stage for a dreaded "internal error". Clipper 5.01 properly flags this as a fatal error while compiling.

Trapping problems via assertions

The BEGIN SEQUENCE block is put to better use when you design it into complex logical blocks right from the start. A technique known as an "assertion" is used to check the success of important processes or the validity of vital assumptions. An example will help explain this technique better than a long-winded technical description. This example checks for valid results the old way, without assertions:

```
result1 := calc1()
if result1 > 0

  result2 := calc2(result1)
  if result2 == "ABC" .and. result1 < 100

    result3 := calc3(result2)
    if result3
      // etc
    endif
  endif
endif
```

In the above example you can imagine how deeply nested the logic can get if a long chain of statements must all meet special criteria. Let's create a new user-defined function to help simplify this common programming problem:

```
function Assert(expr)
if .not. expr
  break
endif
return nil
```

The above function expects as a parameter an expression that returns a logical value. If the expression is false it issues a BREAK. Using Assert(), we can rewrite the previous example.

```
/*
   Check for valid results more efficiently with assertions.
*/
begin sequence
  result1 := calc1()
  Assert(result1 > 0)

  result2 := calc2(result1)
  Assert(result2 == "ABC" .and. result1 < 100)

  result3 := calc3(result2)
  Assert(result3)
end sequence
```

If any of the assertions fails, the BEGIN SEQUENCE block is exited immediately and none of the statements following the failed assertion is executed. The BREAK sends the program to the first statement after END SEQUENCE. The very simple

Assert() function can be made much more useful with the addition of a few features. As discussed previously, blindly leaping out of a block is not very good programming practice. To counter this problem we'll add another parameter that allows us to flag assertion failures:

```
function Assert(expr, flag)
if .not. expr
  flag := .f.
  break
endif
return nil
```

Note that in the next example the flag parameter must be passed by reference via the @ operator. Otherwise the flag value will not be affected by the Assert() function.

```
ok := .t.
begin sequence
  result1 := calc1()
  Assert(result1 > 0, @ok)
  result2 := calc2(result1)
  Assert(result2 == "ABC" .and. result1 < 100, @ok)
  result3 := calc3(result2)
  Assert(result3, @ok)
end sequence
if .not. ok
 ErrorMessage("Calculation Error")
endif
```

The RECOVER statement

So far the techniques we've discussed use features already available in Summer '87. Clipper 5 has added a new twist to BEGIN SEQUENCE via the RECOVER statement. RECOVER is used to create a kind of "else" for the BEGIN SEQUENCE block. Let's return to our Assert() example one more time to illustrate this feature in Listing 27.2. We can eliminate the somewhat pesky flag variable using a RECOVER statement.

Listing 27.2 An improved assertion function which flags failures

```
begin sequence
  result1 := calc1()
  Assert(result1 > 0)
  result2 := calc2()
  Assert(result2 == "ABC")
  result3 := calc3()
  Assert(result3)

recover
  ErrorMessage("Calculation Error")
end sequence

function Assert(expr)
if .not. expr
  break
endif
return nil
```

In the above example if any of the assertions fails the program will branch to the first statement after the RECOVER statement. No need for flags and other special handling. If no BREAK occurs the code in the recovery area is not executed.

The RECOVER statement with the USING clause

Before moving on to the run-time error handling system let's add one more feature to our Assert() function. Sometimes it may be important to know which assertion failed. The only completely foolproof way is to assign a unique flag to each assertion and then test each one within the RECOVER block. Not only does this defeat the purpose of Assert() in the first place, it's also far too much work if all we want is some help when debugging. The RECOVER statement has an option clause called USING. The clause allows the BREAK command to pass information back to the BEGIN SEQUENCE block. The source code below illustrates adding a counter to Assert().

```
function Assert(expr)
static counter := 0
counter++
if .not. expr
  break counter
endif
return nil
```

The counter is incremented every time an assertion is made. When an assertion fails in Assert() the current counter value is passed to the RECOVER USING statement (via the BREAK command in Assert()), as illustrated below. In this example the variable named **which** receives the current value of **counter**, as passed from the BREAK statement.

```
//  This variable will be used later as the recipient of
//  a BREAK value. Assigning a default value is not required.

local which := 0
begin sequence
  Assert(1 == 1)
  Assert(2 == 2)
  Assert(3 == 999)
recover using which
  ? "Assertion failed: "
  ?? which
end sequence
```

In the above example the third assertion will fail so the counter will equal three. Note that the variable used as the target of the USING clause must be defined, just like any other variable. The USING clause does not create it from scratch: You should give it a scope and even a default value if you will be referencing it outside of the BEGIN SEQUENCE block.

One last improvement and we promise to leave this subject alone! The problem with a **static** counter is that it never resets itself to zero. As currently written it will get less and less useful as more assertions are made. Knowing assertion number 7,287 failed isn't very useful if you can't trace the program logic back that far. Let's have the **counter** reset to zero if no parameter is passed to Assert(). This allows us to rely on the **counter** for local recovery strategies. Note that this particular implementation (see listing 27.3) prevents nested sets of BEGIN SEQUENCE because all assertions are using the same counter regardless which block they are called from.

Listing 27.3 A final version of Assert(), this one with the ability to reset the internal counter

```
function Assert(expr)
static counter := 0
if expr == nil
  counter := 0
else
  counter++
  if .not. expr
    break counter
  endif
endif
return nil
```

To reset the assertion **counter** simply call Assert() with no parameter. The example in Listing 27.4 demonstrates two distinct sets of assertions. The failure of the third assertion in the second set results in the assertion **counter** being left at three rather than five.

Listing 27.4 A final version of Assert(), this one with the ability to reset the internal counter

```
local which := 0

Assert() //  Reset assertion counter to zero
begin sequence
  Assert(1 == 1)
  Assert(2 == 2)
recover using which
  ? "First set of assertions failed at: "
  ?? which
end sequence

Assert() //  Reset assertion counter to zero
begin sequence
  Assert("A" == "A")
  Assert("B" == "B")
  Assert("C" == "ZZZ")
recover using which
  ? "Second set of assertions failed at: "
  ?? which
end sequence
```

Armed with a good grasp of BEGIN SEQUENCE..END, BREAK, RECOVER, and USING we can now discuss Clipper 5's wonderful new error handling capabilities.

Error objects and the error block function

The method for handling run-time errors in Clipper 5 touches on several wildly new concepts. It's hard even to introduce the subject without running into unfamiliar jargon and strange new techniques. Let's just jump right in; by the end of the discussion everything should be more clear.

All run-time errors in Clipper 5 are packaged up as an "error object" and passed along to the current error handler. There can be any number of error handlers defined. The most recent one gets first crack at handling the error but can pass the error object on to another handler if one is available. The error object contains all the information known about the error. The error handler contains all the code for dealing with the error. The following "exported instance variables" (as they are referred to in object-oriented jargon) are available from the error object. Detailed descriptions of each will be presented later in this chapter.

args	An array of function or operator arguments.
canDefault	Is default recovery available?
canRetry	Is retry possible after the error?

canSubstitute	Can a new result be substituted after the error?
cargo	Assignable, general-purpose variable.
description	Character description of the error condition.
filename	Name of the file associated with the error.
genCode	Generic Clipper error code number.
operation	Character description of the failed operation.
osCode	Operating system error code number.
severity	Describes the severity of the error on a numeric scale.
subCode	Subsystem-specific error code number.
subSystem	Character description of the subsystem generating the error.
tries	Number of times the failed operation has been attempted.

An error handler refers to these variables to determine its course of action. Each error produces a distinct combination of values. Figures 27.1 through 27.3 show some examples.

Figure 27.1 Example of error object values when a database USE operation fails because the file does not exist

```
use MISSING  // When MISSING.DBF does not exist

args         NIL
canDefault   .T.
canRetry     .T.
canSubstitute .F.
description  Open error
filename     MISSING.DBF
genCode      21
operation    ""
osCode       2
severity     2
subCode      1001
subSystem    DBFNTX
tries        2
```

Figure 27.2 Error object values when an operation fails due to improper arguments

```
? "ABC" + 123  // Attempt to add string and number

args           ABC    123
canDefault     .F.
canRetry       .F.
canSubstitute  .T.
description    Argument error
filename        ""
genCode        1
operation      +
osCode         0
severity       0
subCode        1081
subSystem      BASE
tries          0
```

Figure 27.3 Error object values when a call is made to a non-existent function

```
s := "FooBar()"  // Function does not exist
? &s.

args            NIL
canDefault     .F.
canRetry       .F.
canSubstitute  .T.
description    Undefined function
filename        ""
genCode        12
operation      FOOBAR
osCode         0
severity       2
subCode        1001
subSystem      BASE
tries          0
```

The error object, as we have seen, isn't very complicated at all. But what of the error handler? So far we've mentioned it as something that reacts to the contents of an error object. Where does it come from? What does it do?

Inside ERRORSYS.PRG

Take a look at the top portion of the ERRORSYS.PRG source code file. As in Summer '87, Clipper 5 includes the contents of ERRORSYS.PRG in all of your applications. And as before, you can supply your own ERRORSYS.PRG if you desire. ERRORSYS.PRG contains a small procedure called ErrorSys(), listed below.

```
procedure ErrorSys()
ErrorBlock( { | e | DefError(e) } )
return
```

ErrorSys() is called as the first statement executed when your application starts running. Its job is to establish the first error handler. As you can infer from the source code, should an error occur control will be transferred immediately to a function called DefError(). Let's look more closely at ErrorBlock(). ErrorBlock() expects a code block as a parameter. When an error occurs the error information is collected and placed in an error object, then the specified code block will be passed the error object as a parameter. Under most circumstances the error handling code block merely passes the error object along to a user-defined function. The user-defined function is then responsible for taking action and reacting to the error. What does an error handling function look like?

The standard error handler

Reprinted here for more convenient reference is the source code for the standard error handler that comes with Clipper 5.01. The source code is found in ERRORSYS.PRG. This code forms DefError(), which ErrorSys() establishes as the function the code block will call via the error block defined with ErrorBlock(). That's a mouthful! Keep thinking about it until you get it straight. Here's a step-by-step description of how Clipper handles errors by default.

1. When Clipper starts executing your application it first calls the function named ErrorSys().

2. ErrorSys() uses the ErrorBlock() function to “post” an error handling code block.

3. The code block receives an error object as a parameter when an error occurs. The code block immediately calls DefError() with the error object as a parameter.

4. The DefError() function actually does all the error handling, based on the contents of the error object it received as a parameter from ErrorSys().

The DefError() source code (Listing 27.5) makes use of the manifest constants defined in ERROR.CH. We’ll talk more about ERROR.CH later in this discussion.

Listing 27.5 Source code for Clipper’s DefError() function, located in ERRORSYS.PRG

```
/* Copyright (c) 1990 Nantucket Corp. All rights reserved.
(Reprinted here for reference. The format has been edited slightly.)
*/


static func DefError(e)
local i, cMessage, aOptions, nChoice

// by default, division by zero yields zero
if ( e:genCode == EG_ZERODIV )
  return (0)
endif

// for network open error, set NETERR() and subsystem default
if ( e:genCode == EG_OPEN .and.;
     e:osCode == 32 .and. e:canDefault )
  NetErr(.t.)
  return (.f.)
endif
```

```
   // for lock error during APPEND BLANK, set NETERR()
   // and subsystem default

   if ( e:genCode == EG_APPENDLOCK .and. e:canDefault )
     NetErr(.t.)
     return (.f.)
   endif

   // build error message
   cMessage := ErrorMessage(e)

   // build options array
   // aOptions := {"Break", "Quit"}
   aOptions := {"Quit"}

   if (e:canRetry)
    aadd(aOptions, "Retry")
   endif

   if (e:canDefault)
    AAdd(aOptions, "Default")
   endif

   // put up alert box
   nChoice := 0
   do while ( nChoice == 0 )

     if ( Empty(e:osCode) )
       nChoice := Alert( cMessage, aOptions )
     else
       nChoice := Alert( cMessage + ;
              ";(DOS Error " + NTRIM(e:osCode) + ")", ;
               aOptions )
     endif

     if ( nChoice == NIL )
       exit
     endif
  enddo
```

```
if ( !Empty(nChoice) )
  // do as instructed
  if ( aOptions[nChoice] == "Break" )
    Break(e)

  elseif ( aOptions[nChoice] == "Retry" )
    return (.t.)

  elseif ( aOptions[nChoice] == "Default" )
    return (.f.)
  endif
endif

// display message and traceback
if ( !Empty(e:osCode) )
  cMessage += " (DOS Error " + NTRIM(e:osCode) + ") "
endif

? cMessage
i := 2
do while ( !Empty(ProcName(i)) )
  ? "Called from", Trim(ProcName(i)) + ;
    "(" + NTRIM(ProcLine(i)) + ")  "
  i++
enddo

// give up
ErrorLevel(1)
QUIT

return (.f.)
```

Note how every aspect of the way Clipper handles a run-time error is programmed using Clipper statements and functions. All the handling assumptions, like dividing by zero and network errors, are laid out clearly in the DefError() function. Of particular interest is the "traceback" loop. Note how easy it is to document the current calling sequence for the entire application.

Take a look at the source code in Listing 27.6. If you read it carefully you can see each element of a Clipper run-time error message being built. This function, ErrorMessage(), is found in ERRORSYS.PRG and is used by DefError() to construct the run-time error messages. The function has been declared static, so only DefError() (and other functions within the ERRORSYS.PRG source code file) can call it. If you want to use the function in your own routines you'll have to snag a copy and incorporate separately.

Listing 27.6 Source code for Clipper's ErrorMessage() function, located in ERRORSYS.PRG, reprinted here for reference

```
/* Copyright (c) 1990 Nantucket Corp.  All rights reserved. */
static func ErrorMessage(e)
local cMessage

// start error message
cMessage := if( e:severity > ES_WARNING, "Error ", "Warning " )

// add subsystem name if available
if ( ValType(e:subsystem) == "C" )
  cMessage += e:subsystem()
else
  cMessage += "???"
endif

// add subsystem's error code if available
if ( ValType(e:subCode) == "N" )
  cMessage += ("/" + NTRIM(e:subCode))
else
  cMessage += "/???"
endif

// add error description if available
if ( ValType(e:description) == "C" )
  cMessage += ("  " + e:description)
endif
```

```
// add either filename or operation
if ( !Empty(e:filename) )
  cMessage += (": " + e:filename)
elseif ( !Empty(e:operation) )
  cMessage += (": " + e:operation)
endif

return (cMessage)
```

If you want to alter the way Clipper handles run-time errors you could modify DefError() or even change the code block in ErrorSys(). However, you'd be back to the Summer '87 problem of needing to handle special cases by adding more code to the standard error handler or by calling special-purpose functions from inside the standard error handler. This is messy and creates maintenance problems. Clipper 5 provides us with a much better alternative, which we will now discuss.

Run-time Warnings

Clipper 5.01 adds an interesting new kind of error handling, the run-time warning. Presently only one such warning is available, concerning a low memory situation. Hopefully this trend will continue and many more such warnings will be provided. These warnings are exactly that, just a warning and not an outright error condition. The Clipper run-time engine is informing you of a situation that potentially could cause real errors. Examples of other such warnings are low disk space and keyboard buffer overflows— things you may or may not be concerned about, depending on the situation. The usual default action is to ignore the warning. However, you can install your own error handler that instead pops-up a warning box or takes other actions.

A chain of error handlers

As described previously, ErrorBlock() is used to "post" a code block as the function to call when an error occurs. You can post as many code blocks as often as you wish. The ErrorSys() function posted DefError() as the first error handler when your application started running. You can replace it with a different error handler just as easily.

```
errorblock( { | errorObject |  NewErrorHandler(errorObject) } )
```

From this point on, all error objects will be sent to NewErrorHandler() rather than DefError(). It's unlikely that you will want to completely replace the entire error handling system. It is more appropriate to establish a special error handler to deal with a specific problem. If the error turns out to be something other than the one your handler is prepared to deal with you simply pass the error to the main error handling routine. This "pass it along" technique is possible because ErrorBlock() returns the error block that currently exists. You save the current error block, post your special version, and restore the original one when the special one is no longer needed.

Listing 27.7 gives an example. We want to handle a missing database on our own and pass all other errors on to the main error handler, DefError(). Furthermore, we want to handle missing databases only in a small section of code, not for the entire application. Remember that DefError() is already posted as the main error handler, and that the default way to handle missing databases is to report the problem and exit to DOS.

Listing 27.7 An alternative error handler

```
/*
   oldErr will hold the current error handler.
   Errorblock() posts MissingDBF() as the new error handler.
*/
oldErr := errorblock()
errorblock( { | e | MissingDBF(e, oldErr) } )

/*
   We must delimit the section of code where we want
   to use our own local error handler. If the MISSING.DBF
   file does not exist the MissingDBF() function will be
   called as the error handler.
*/
begin sequence
```

```
  use VENDOR new
  use INVOICE new
  use INVENT new
  use MISSING new
end sequence

/*
   We're beyond the section where we want to handle
   missing databases on our own, so we restore the original
   error handler.
*/
errorblock(oldErr)

/*
   If this file doesn't exist the standard error handler
   will deal with the error.
*/
use ANOTHER new

return nil
```

Note how we can open a large number of databases in rapid succession without having to check for the existence of each one individually. Without a local error handler we would need a long series like the following.

```
err := .f.
if file("VENDOR.DBF")
  use VENDOR new
else
  err := .t.
endif
// etc for the other files: INVOICE, INVENT and MISSING

if err
 @ 0,0 say "Error: All database files were not opened."
endif
```

Listing 27.8 contains the source code for the MissingDBF() function that we posted as a local error handler.

Listing 27.8 An error handler for dealing with missing databases

```
function MissingDBF(errObj, origBlock)
/*
   Special-purpose error handler for missing databases.
*/
#include "ERROR.CH"
local result

/*
  Check to see if the error is an Open Error.
  If so, report the error and BREAK back to the
  local section of code.
*/
if errObj:genCode == EG_OPEN
 @ 0,0 say "Error: All database files were not opened."
 break
/*
  Error isn't what this function is designed to handle, so
  pass the error along to the original error handler.
  Before passing it along we use the cargo instance variable
  to tack on a reminder of where the error has been. This
  might be useful in debugging. If several error handlers pass
  along the error they can all tack on a message to the cargo
  string.

*/
else
 if valtype(errObj:cargo) <> "C"
   errObj:cargo := ""
 endif
 errObj:cargo += "(Passed along from MissingDBF)"
 result := eval(origBlock, errObj)
endif

return result
```

The code for your error handler should be tested thoroughly because a run-time error within the error handler will likely result in recursive calls to the error handler. Eventually Clipper will run out of memory and crash to DOS (if you are lucky!).

We can take advantage of the RECOVER USING statement to pass the error object back to the local error handler for use in a local attempt at recovery. If we substitute the break in MissingDBF() with "break errObj", like so:

```
if errObj:genCode == EG_OPEN
  @ 0,0 say "Error: All database files were not opened."
  break errObj
```

Then we can do something like the following.

```
#define OS_ERR_NOTFOUND  2
#define OS_ERR_HANDLES   4
local the_error

  begin sequence
    use ONE new
    use TWO new
    use THREE new

  recover using the_error
    if the_error:osCode == OS_ERR_HANDLES
      @ 1,0 say "Too many files open."
    elseif the_error:osCode == OS_ERR_NOTFOUND
      @ 1,0 say "File not found."
    endif
  end sequence
```

When MissingDBF() issues a BREAK it passes the object containing the error to the variable specified by USING, in this case, **the_error**. You can then use the **the_error** object to determine more specifically what happened to cause the error condition.

The ERROR.CH file

To make error handling code easier to follow and able to withstand future additions and modifications to the error object class, the code in ERRORSYS.PRG and our examples in this chapter make references to manifest constants defined in ERROR.CH (see Figure 27.4). You should always refer to the manifest constants and never to the actual error code numbers. The numbers may change in future versions of Clipper. If you stick to the manifest constants you'll need only to recompile your error handler functions and the codes will be adjusted.

Figure 27.4 The contents of ERROR.CH reprinted for more convenient reference

```
/* Copyright (c) 1990 Nantucket Corp. All rights reserved */

// Severity levels (e:severity)
#define ES_WHOCARES         0
#define ES_WARNING          1
#define ES_ERROR            2
#define ES_CATASTROPHIC     3

// Generic error codes (e:genCode)
#define EG_ARG              1
#define EG_BOUND            2
#define EG_STROVERFLOW      3
#define EG_NUMOVERFLOW      4
#define EG_ZERODIV          5
#define EG_NUMERR           6
#define EG_SYNTAX           7
#define EG_COMPLEXITY       8

#define EG_MEM             11
#define EG_NOFUNC          12
#define EG_NOMETHOD        13
#define EG_NOVAR           14
#define EG_NOALIAS         15
#define EG_NOVARMETHOD     16

#define EG_CREATE         20
#define EG_OPEN            21
```

```
#define EG_CLOSE            22
#define EG_READ             23
#define EG_WRITE            24
#define EG_PRINT           25

#define EG_UNSUPPORTED      30
#define EG_LIMIT            31
#define EG_CORRUPTION       32
#define EG_DATATYPE         33
#define EG_DATAWIDTH        34
#define EG_NOTABLE          35
#define EG_NOORDER          36
#define EG_SHARED           37
#define EG_UNLOCKED         38
#define EG_READONLY         39
#define EG_APPENDLOCK       40
```

The error object in detail

Now that we've been introduced to all the pieces of Clipper 5's error handling strategy it's time to discuss the exported instance variables supplied by an error object. Each variable has a distinct role to play in helping you understand and possibly recover from errors. (For the purpose of keeping the instance variables straight we're assuming an error object named "e" is being evaluated in the following discussions.)

e:args

e:args is an array of function or operator arguments and is not always used. **e:args** is undefined (NIL) if not used.

e:canDefault

e:canDefault indicates whether or not default recovery is available. You request default recovery by returning **.F.** from the error handling function. Exactly what happens when default recovery is requested depends on the subsystem (see **e:genCode**,below) that was responsible for the error.

e:canRetry

e:canRetry indicates whether or not it's possible to retry the operation that caused the error. You request a retry by returning **.T.** from the error handling function. As with **e:canDefault**, the ability to retry a failed operation depends on the subsystem that generated the error.

e:canSubstitute

e:canSubstitute indicates whether or not a different result can be substituted for the error. You make a substitution by returning the value from the error handling function. Because of this, **e:canSubstitute** is not an option when either **e:canDefault** or **e:canRetry** is true.

e:cargo

e:cargo is a variable you can assign on your own, useful for recording information about the error. In this chapter, the MissingDBF() function uses **e:cargo** to pass along a note that it received the error but passed it back to the error handler that preceded it. (See Listing 27.8 for an example use of **e:cargo**.)

e:description

e:description is a description of the error condition, based on the generic error code, **e:genCode**, or supplied by the subsystem if **e:genCode** is zero. Not all subsystems produce error descriptions.

e:filename

e:filename is the name of the file associated with the error. If the **e:filename** is an empty string it means that either the operation doesn't use a filename or the subsystem doesn't keep filenames.

e:genCode

e:genCode is the generic Clipper error code number. See ERROR.CH for a list of codes. If **e:genCode** is zero it means the error is unique to the subsystem and no generic code has been assigned. Generic codes allow different subsystems to share the same error strategy, for example, an "open error" is applicable to .DBF files as well as SET ALTERNATE TO.

e:operation

e:operation is a description of the failed operation. Operations are function names, like STR() or VAL(), or operators like **+** or **-**. Undefined functions or variables are also reported via **e:operations**.

e:osCode

e:osCode is the operating system's error code number, if one is applicable to the error, otherwise **e:osCode** will be zero. **e:osCode** and DOSERROR() report the same number, and if **e:osCode** is changed DOSERROR() will also get the new value.

e:severity

e:severity describes the severity of the error on a numeric scale. See ERROR.CH for manifest constants that associate descriptions with the scale. The levels of severity have the following interpretations.

Who Cares: The error condition was raised to supply information, it isn't really an error.

Warning: The error condition isn't fatal...yet. Operations may continue but a more serious error could occur in the future.

Error: This is an honest-to-goodness error, meaning the operation did not succeed. Something has to be corrected before operations can safely continue.

Catastrophic: The error condition is serious enough that operations must halt immediately and the application is terminated.

Presently Clipper will only generate errors with severity levels of **Warning** or **Error**. The other severity levels are likely to be used by add-on product vendors, especially those for replaceable database drivers (RDDs). Nantucket may also use the other extremes in future releases of Clipper.

e:subCode

e:subCode is a subsystem-specific error code number. Every error gets a unique subsystem error code. If zero it means the subsystem does not assign error codes.

e:subSystem

e:subSystem is a description of the subsystem that generated the error. Clipper functions and operators are part of the "BASE" subsystem. Errors generated by the database driver will contain the name of the database driver. If you are using the default Clipper database driver, **e:subsystem** will be "DBFNTX". Future database drivers (known as RDDs or replaceable database drivers) will supply different subsystem names and subsystem error codes.

e:tries

e:tries is a counter that indicates the number of times the failed operation has been attempted. Not all subsystems track this number so it may be zero. You can use **e:tries** to limit the number of additional attempts that are made.

Error object inspection and recording

As an aid in identifying what different error objects contain, this book includes a function, ErrorSaver(), that can be installed as a replacement to DefError(). It displays all information about the error, including memory status, DOS information, and procedure traceback, and then optionally appends a copy to a text file. You can use the results for learning about different types of errors and developing strategies to recover from them or even avoid them in the first place.

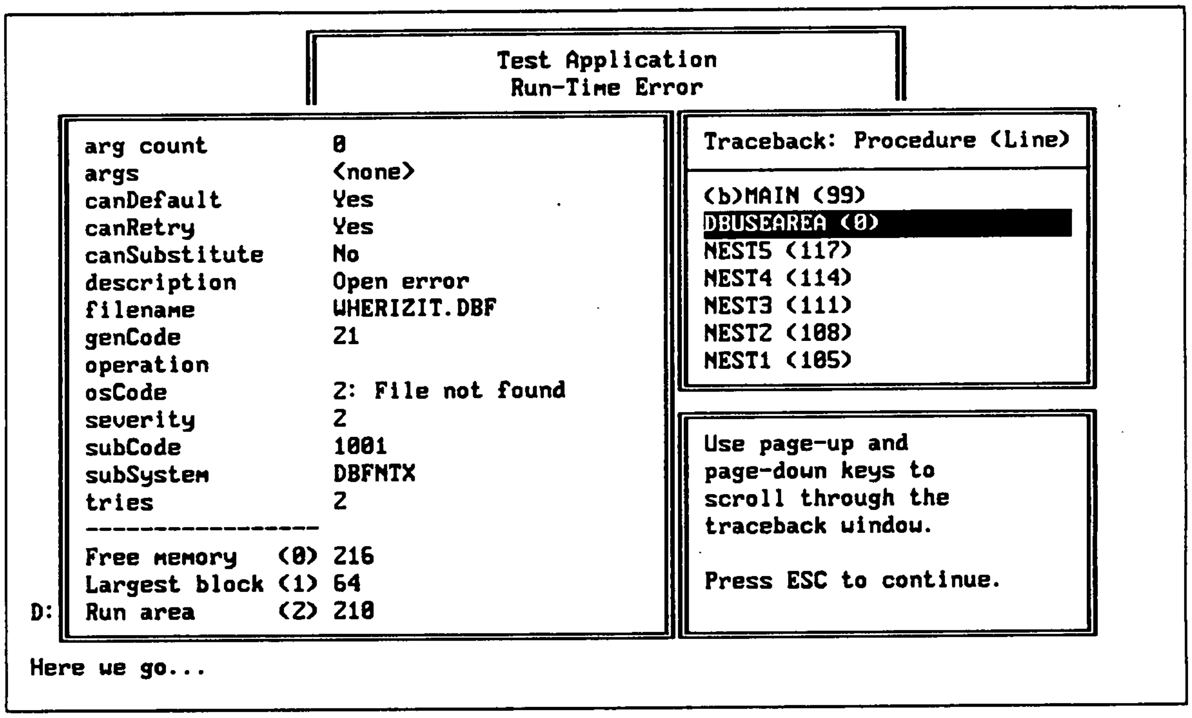
After application development is complete, ErrorSaver() is still useful for keeping an accurate log of all run-time errors rather than relying on the end-user to print screens or copy down vital information.

To install this function as the default error handler all you need to do is add the following as the first line of code in your application. It specifies that errors will be recorded in a file called ERR.TXT. You can omit the filename parameter if you don't want the information recorded.

```
errorblock( { |e| ErrorSaver(e, "ERR.TXT") } )
```

Figure 27.5 shows the ErrorSaver() screen in action, and Figure 27.6, the resulting error log file entry.

Figure 27.5. A sample ErrorSaver() screen



```
                    Test Application
                    Run-Time Error

arg count           0                    Traceback: Procedure (Line)
args                <none>
canDefault          Yes                  (b)MAIN (99)
canRetry            Yes                  DBUSEAREA (0)
canSubstitute       No                   NEST5 (117)
description         Open error           NEST4 (114)
filename            WHERIZIT.DBF         NEST3 (111)
genCode             21                   NEST2 (108)
operation                                NEST1 (105)
osCode              2: File not found
severity            2                    Use page-up and
subCode             1001                 page-down keys to
subSystem           DBFNTX               scroll through the
tries               2                    traceback window.
-----------------
Free memory    (0) 216                   Press ESC to continue.
Largest block  (1) 64
D: Run area    (2) 210

Here we go...
```

Figure 27.6 A sample entry in the error log file.

```
================================================================
ErrorSaver: This run-time error logged on 03/05/91 at 15:10:30
Application: Test Application
Operating system = DOS 3.20, network = <none>
Available diskspace = 1,349,632 in C:\TEST
PATH  = c:\clipper5\bin;c:\dos;c:\norton;c:\utils;c:\bat
COMSPEC = C:\COMMAND.COM
CLIPPER = F51
________________________________________

Traceback: Proc (Line)  Error Information
~~~~~~~~~~~~~~~~~~~~~~  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   WUSE (0)               arg count         0
   NEST5 (114)            args              <none>
   NEST4 (111)            canDefault        Yes
   NEST3 (108)            canRetry          Yes
   NEST2 (105)            canSubstitute     No
   NEST1 (102)            description       Open error
   MAIN (98)              filename          WHERIZIT.DBF
                          genCode           21
                          operation
                          osCode            2: File not found
                          severity          2
                          subCode           1001
                          subSystem         DBFNTX
                          tries             2
                          __________
                          Free memory   (0) 185
                          Largest block (1) 64
                          Run area      (2) 178
```

See Appendix B for the complete ERRSAVE.PRG source code listing.

Functions related to error handling

Clipper 5 provides a variety of functions used in error detection and handling. We've covered several of them already. Most are quite simple but are still vitally important to a comprehensive error handling strategy. The following list can serve as a reference when searching for tools in your error detection and recovery programming.

Alert()	A general-purpose user-interface function used in ERRORSYS.PRG to report errors and optionally solicit a response from the user. You can call Alert() for your own purposes; it can be used outside of ERRORSYS.PRG.
DefError()	The default error handler. Found in ERRORSYS.PRG, it can be altered or, preferably, superseded by another call to ErrorBlock() that installs a new error handler.
DosError()	The error number of the most recent DOS-related error. Zero indicates no error. Keep in mind that DosError() can be assigned a new value, which allows your error handler to decide if the original error number (or even zero) should be returned to the main application.
ErrorBlock()	Returns current error handling code block and optionally posts a new one.
ErrorLevel()	Returns current DOS ERRORLEVEL number and optionally assigns a new value.
ErrorSys()	Always called as first executed statement when your application starts running. Found in ERRORSYS.PRG, it can be altered or, preferably, superseded by another call to ErrorBlock() that installs a different error handler.

Ferror()	Result of most recent low-level file operation. Returns 0 if the operation was successful or a positive number if an error occurred.
NetErr()	Returns result of most recent USE or APPEND BLANK command: true if it succeeded, false if it failed. Keep in mind that NetErr(), like DosError(), can be assigned a new value, which allows your error handler to decide if it should be true or not when returned to the main application.

Alert() in more detail

The Alert() function is an important addition to your programming tools and should not be associated only with runtime errors. First, Alert() can detect the absence of Clipper's full-screen terminal I/O and reverts to standard TTY-style I/O. TTY-style means the lowest common denominator among video adapter cards and monitors, allowing only simple output capabilities. This is essential for having a generic error-handling system that does not depend in any way on the functions linked into an application. Using Alert() allows you to develop routines that are more device-independent than regular Clipper routines using MENU TO and @..SAY. Second, Alert() packs a very powerful pop-up message box and menu selection routine into a simple function call.

Alert() accepts two parameters, a message and an array of options. It returns the number of the option selected or zero if no selection was made.

```
nChoice := alert(cMessage [, aOptions])
```

If an array of options is not passed, Alert() displays a single "OK" choice. The message string accommodates multiple-line messages by using the semicolon as a line separator. The following example produces a box with the two message lines centered above the three options.

```
now_what := alert("Something terrible;has happened!", ;
                  {"Abort","Retry","Ignore"})
```

You are free to use Alert() in your own error handling routines as well as anywhere else in your application you see fit.

Note that Alert() is actually a preprocessor translation directive. See the file ASSERT.CH for more details. As of this writing, ASSERT.CH can be found in the \CLIPPER5\SOURCE\SAMPLE subdirectory, and not in \CLIPPER5\INCLUDE as you might expect. This may change in subsequent releases.

Creating your own errors

Who needs help creating errors? You say you do fine on your own, thank you? The error class, like all object classes, has a constructor function that returns an error object. You can use this newly created object to produce run-time errors that are difficult (and often tedious) to generate "naturally". Who wants to fill up a hard disk or trash DOS's memory control blocks just to see if an error handler works correctly for that situation? The error doesn't really occur, but the error handler doesn't know that. This also means that error recovery operations may not work correctly since the error condition doesn't really exist, so it's not perfect.

To create a new error object you simply call ErrorNew(), of all things, and receive one.

```
test := errorNew()
```

Then you can construct any nasty errors you can think of by simply making assignments to the object's instance variables. The following sets up a "file not found" situation for the USE command.

```
test:filename := "TEST.DBF"
test:genCode  := EG_OPEN
test:osCode   := 2
subCode       := 1001
subSystem     := "DBF"
```

Don't bother assigning a description if you use **e:genCode**, because as soon as you assign the **e:genCode** variable the associated description is placed in the description variable. Once the error is defined as much as you need it to be for testing, pass it as a parameter to an error handling function.

```
DefError(test)
```

The error handler will react to the error object as if the error it represents had actually happened. Feels nice to be in the driver's seat of an error condition for a change, doesn't it? Go ahead, torture DefError() all you want.

Note: Since DefError() is by default a static function, in order to call it directly you'll have to link in a copy of ERRORSYS.PRG and either make your call from within that source code file or edit the function definition to be non-static. This is not a limitation inherent with DefError(), it's just the way all static functions work.

Summary

Error detection, reporting and recovery have been made considerably more powerful, and at the same time, more easy to implement in Clipper 5. You've seen how the BEGIN SEQUENCE block, coupled with BREAK and RECOVER USING, can be used to construct bullet-proof program logic. We tackled the error object class and found it to be very simple to understand. You know how ERRORSYS.PRG works and how to install our own error handlers. Now all you need to do is put all this knowledge to work keeping your applications running smoothly even when errors occur.

CHAPTER 28

Obsolete Commands and Functions

Change is inevitable in anything related to computers, and the evolution of Clipper is no exception. Up through the release of Clipper 5 Nantucket followed an often twisting path, managing to stay relatively compatible with other DBASE dialects while at the same time extending the language in powerful and unique ways. Clipper 5 is the first release that takes a more dramatic departure from the existing standards. On the whole, this departure makes Clipper a better, more stable platform for software developers and signals Nantucket's intention to make Clipper a distinct alternative to other languages, not just an alternative to the other DBASE dialects.

Not surprisingly, to accomplish this transformation Clipper programmers are going to have to alter their styles and techniques to take advantage of what future versions of Clipper will offer. Clipper 5 can be treated to a certain extent like an improved Summer '87; you will likely see some improvements in speed and memory management simply by recompiling existing applications and linking them with RTLink. However, an attitude of "it's the same, but better" isn't going to get you very far. In fact, Nantucket has made some very broad hints about the direction in which they are headed. These hints take the form of a notation in the Clipper documentation regarding certain "compatibility" commands and functions. All such commands and functions are indicated with an asterisk next to their entries in the printed documentation and in the Norton Guides. The book you are reading has taken these notations to heart and consciously avoids the commands, functions, and techniques that don't fit with Nantucket's vision of Clipper's future. We think you'll agree that in all cases a switch from the old to the new will improve your programming technique and produce better, more reliable applications. If you do not take heed and alter your techniques you may find it difficult to work with future versions of Clipper.

This chapter is dedicated to identifying these commands and functions, explaining their deficiencies and offering reasonable alternatives. By following the advice offered here you'll be in a better position to make a smooth transition to Clipper 5 as well as future versions of Clipper.

Obsolete statements

Statements form the fundamental control features of Clipper. They are used to divide your applications into modules, to indicate the storage class of variables, and to control logical flow at run-time. There are a few new statements that are covered in detail in other chapters of this book: **field**, **local**, **memvar** and **static**. There are two that have been marked as obsolete: DECLARE and DO. There are four that aren't considered obsolete but should, in our opinion, be avoided to make best use of the language: **parameters**, **private**, **procedure**, and **public**.

DECLARE

DECLARE is used to create **private** arrays and variables.

Deficiency: Because it has the exact same effect as the **private** statement, DECLARE is not required under any circumstances.

Remedy: You can replace all occurrences of DECLARE with **private** without causing any side-effects. However, we recommend avoiding **private** as well. See Chapter 6, "Variable Scoping," for details.

DO

DO is used to invoke a procedure. When used with parameters, DO passes them by reference instead of by value.

Deficiency: Clipper functions can be called on lines by themselves, so they function identically to commands in that respect.

```
do MyRoutine  //  These have the identical effect
MyRoutine()   //  of executing "MyRoutine."
```

The most troublesome feature of commands is that they pass parameters by reference, meaning the values of any variables used as parameters can be altered by the procedure. This can cause unintended side-effects and reduces the reliability of your applications.

```
x := 123
y := 789
do MyProc with x
MyFunc(y)
? x                     //  "abc", the value was altered
? y                     //  789, the value was not altered

procedure MyProc
parameter p
p := "abc"
return

function MyFunc
parameter p
p := "xyz"
return nil
```

Remedy: Design all new routines as functions and stop using procedures. Edit existing procedures into functions in situations where it's an easy change; however, you may introduce errors in complex applications by blindly converting every procedure to a function.

If you need to take advantage of a procedure's ability to directly alter the contents of its parameters, use the "pass by reference" operator, @, instead. The @ operator makes it obvious to anyone reading your source code that you are passing the parameter by reference and they should anticipate potential changes to the value. Use of @ also has the desirable effect of allowing you to pass some parameters by reference and others by value, whereas with a procedure all parameters are passed by reference. See Chapter 5, "Operators", for a detailed discussion of passing variables by reference.

```
x := 123
y := 789
MyFunc(@x, y)
? x               // "abc"
? y               // 789

function MyFunc
parameters a, b
a := "abc"
b := "xyz"
return nil
```

If you have a large number of procedures in existing applications but still want to eliminate the potential side effects of passing parameters by reference, note that by simply enclosing each parameter in parentheses you make it an *expression.*

```
// The value of x will remain unchanged
// no matter what happens in the procedure
x := 123
do MyProc with (x)
```

Many Clipper programmers use procedures to indicate that the routine is called for the effect it produces and not for any value it might return. While this is a reasonable use for procedures, the same effect can be gained with a preprocessor #xcommand directive.

```
#xcommand do MyProc  =>  MyProc()
```

Although all functions return values, you do not have to acknowledge the value when the function is called. We recommend returning NIL when the function is not designed to return a useful value. Some programmers use the C language convention of using the keyword VOID to indicate a function has no return value.

```
#define VOID nil

function Nothing
return VOID
```

PROCEDURE

PROCEDURE is used to mark the start of a routine.

Deficiency: The PROCEDURE statement is tied to the DO statement, which was discussed in detail in the previous section.

Remedy: Try to use FUNCTION exclusively.

PRIVATE and PUBLIC

The PRIVATE and PUBLIC statements create and declare the scope of variables.

Deficiency: The variables are dangerously visible to too much of the rest of the application

Remedy: PRIVATE and PUBLIC should be replaced with LOCAL and STATIC when possible. See Chapter 6, "Variable Scoping," for details.

PARAMETERS

The PARAMETERS statement establishes a list of variable names used as parameters within procedures and functions.

Deficiency: The PARAMETERS statement creates **private** variables.

Remedy: Parameters should be defined via syntax that creates local variables. Again, see Chapter 6 for details.

```
function One
parameters a, b, c
  //  a, b, and c are PRIVATE variables
return nil

function Two(x, y, z)
  //  x, y and z are local variables
return nil
```

Note that when you make the switch to LOCAL parameters you will no longer be able to use the TYPE() function to test them. You must instead us VALTYPE(). See Chapter 6 for a detailed discussion of TYPE() and VALTYPE().

Obsolete commands

Although it isn't apparent unless you study the contents of STD.CH — the main preprocessor header file — Clipper 5 doesn't really have any commands! Strange but true, the language has been divided into statements, functions, and preprocessor directives, with no commands. What we traditionally consider commands, like GOTO TOP and READ, are actually converted to function calls by the preprocessor. However, though implemented differently, commands are still an important part of the language. Using commands has always made the dBASE dialects easier to understand and more accessible for novices than other languages like C or Pascal. This remains true in Clipper 5, and in fact it's now possible to write your own commands via the incredible preprocessor.

With that said, there are some commands which have outlived their usefulness in Clipper. Some probably never should have made it as far as Summer '87, much less Clipper 5, and others are only now made obsolete by new capabilities. Either way, the sooner you leave them behind, the better. However, with the advent of user-defined commands it is now possible to recover from future changes by adding commands you need right back into the language.

CALL

CALL executes procedures external to your Clipper application, written in C or assembler.

Deficiency: CALL is a crude way to execute external procedures and does not lend itself to seamless integration with the rest of the Clipper language. It exists primarily for compatibility with routines originally meant for use with dBASE III Plus.

Remedy: Clipper's Extend System is a considerably more elegant and efficient way to accomplish the same thing.

CANCEL

CANCEL stops program execution, closes all open files and exits to the operating system.

Deficiency: CANCEL is a synonym for QUIT and has the exact same effect. CANCEL in interpreted dialects exits only to the "dot prompt", which is unavailable in Clipper.

Remedy: Replace all occurrences of CANCEL with QUIT, there are no side-effects.

CLEAR ALL

CLEAR ALL releases all **private** and **public** variables and closes all open files.

Deficiency: This command has effects more wide ranging than is considered appropriate in an application while it is executing. Nantucket sometimes refers to CLEAR ALL as "the command that knew too much."

Remedy: CLEAR ALL should be replaced with a combination of other CLEAR commands that have the desired effect. CLEAR ALL (as well as CLEAR MEMORY) has no effect on **local** and **static** variables, so if you follow our advice regarding avoiding the use of **private** and **public** variables, CLEAR ALL will have no more effect than CLOSE ALL. With that said, CLEAR ALL isn't that big of a deal and can be easily recreated with a user-defined command.

DIR

DIR displays a directory listing.

Deficiency: In this the age of modern user interfaces and careful control of the screen, DIR is an anachronism which lobs text to the screen in an almost uncontrolled manner.

Remedy: The awkward DIR should be replaced by a call to DIRECTORY() to load the desired directory information into an array, followed by a simple array-based TBrowse that will allow the user to view the results. The only side effect is the addition of a DIR replacement function and a new user-defined command that converts the bumbling DIR to the new function call.

```
#command DIR [<filespec>] => DirBrowse([<filespec>])

function DirBrowse(spec)
  //  Emulation of DIR's output in a TBrowse.
return nil
```

FIND

FIND searches the current index for a matching key value.

Deficiency: FIND was originally intended for use with literal values at a "dot prompt" command line. Clipper has no such prompt and is almost always working with variable names and expressions, rendering FIND almost useless.

Remedy: FIND can be replaced with SEEK with no side effects as long as the search key is made into an expression.

```
find MN
seek "MN"   //  Same effect as FIND MN.
seek MN     //  Assumes a variable called MN is available,
            //  will likely cause a run-time error.
```

NOTE

Note makes a single-line comment in a source code file.

Deficiency: As if there weren't enough options for comments. You have //, *, &&, and /* */. The use of NOTE is just begging to be confused with real commands that actually do something.

Remedy: Replace all occurrences of NOTE with //. If you really prefer to do all that typing you can always create a user-defined command.

```
#command COMMENTARY <*x*> =>
#command DO NOT COMPILE THIS <*x*> =>
```

SAVE and RESTORE SCREEN

SAVE SCREEN and RESTORE SCREEN are used to save a copy of the current screen and then restore it.

Deficiency: These commands work on the entire screen, and can even be used to save and restore without specifying a variable to hold the screen image. There are better ways to handle screen saves, and you should always store the screens explicitly to a variable name to prevent collisions with other functions that also need to save or restore the screen.

Remedy: Replace SAVE SCREEN and RESTORE SCREEN with calls to the SAVESCREEN() and RESTSCREEN() functions, which can operate on partial screens and must be used with a variable name.

```
//  These are equivalent, entire screen is stored to TEMP.
```

```
save screen to temp
temp := savescreen(0,0, maxRow(), maxCol())

// These are also equivalent, entire screen is restored from
// TEMP.
restore screen from temp
restscreen(0,0, maxRow(), maxCol(), temp)
```

Admittedly, when you simply want to save and later restore the entire screen with a minimum of fuss, using the verbose SAVESCREEN() and RESTSCREEN() functions is a bit of a pain. The following reduces the process to a single function call.

```
temp := ScreenShot()
//
//  Do something that messes up the screen.
//
ScreenShot(temp)  // Screen is now restored.
quit

function ScreenShot(scr)
/*
  Return a screen save, or restore from screen passed as parameter.
*/
if scr == nil
  scr := savescreen(0,0,maxRow(),maxCol())
else
  restscreen(0,0,maxRow(),maxCol(), scr)
endif
return scr
```

SET COLOR

The SET COLOR command establishes screen colors.

Deficiency: This command blindly sets the colors and offers no way to save the current color settings.

Remedy: SET COLOR can be completely replaced by the SETCOLOR() function. SETCOLOR() returns the current color setting string before establishing the new string. Saving the existing color string allows you to restore the original colors if necessary. This is an important consideration when writing general-purpose functions.

```
setcolor("G/N")
@ 10, 15 say "Status: "
if ferror()
  Warning("There's an Error!")
endif
@ 12, 15 say "Still using G/N color."
quit

function Warning(message)
local clr
// Flashing red on white for a warning message.
clr := setcolor("R*/W")
@ row(), col() say message
// Restore colors to what they were previously.
setcolor(clr)
return nil
```

When it comes right down to it, even the use of SETCOLOR() as demonstrated above can be eliminated via the new COLOR clause available with @...SAY and other commands. This example should serve only to illustrate the use of SETCOLOR() to save and restore the current color.

SET EXACT

SET EXACT specifies whether character strings must match exactly when comparing them.

Deficiency: SET EXACT is an application-wide setting and may have far-reaching consequences that you do not intend.

Remedy: Write expressions that yield precisely the type of match you wish to make, don't rely on the setting of SET EXACT. The double equal (==) operator is better used to compare two strings for an exact match.

```
? "abc" = "a"    //  True if EXACT is OFF, False if ON.
? "abc" == "a"   //  False no matter what the status of EXACT.
```

See Chapter 20, Tables 20-1, and 20-2 for the the rules of using SET EXACT.

SET EXCLUSIVE

SET EXCLUSIVE specifies whether all database files should be opened as shared or as exclusive.

Deficiency: SET EXCLUSIVE is an application-wide setting that is better handled on a file-by-file basis.

Remedy: Use the EXCLUSIVE and SHARED options of the USE command rather than relying on the status of SET EXCLUSIVE. The programmer should make an explicit reference to EXCLUSIVE when opening a database file. This eliminates toggling SET EXCLUSIVE ON and OFF all over your application and the resulting potential for mistakes.

```
use MYFILE exclusive
use OURFILE shared
```

SET FORMAT

SET FORMAT specifies a format file to be used when a READ is executed.

Deficiency: Clipper doesn't really use format files the way they are implemented in interpreted dBASE dialects. Clipper treats SET FORMAT TO **<name>** the same as DO **<name>**. Even when implemented as intended, SET FORMAT TO is pretty useless in the other dBASE dialects.

Remedy: Eliminate SET FORMAT TO by calling an equivalent function that displays prompts and establishes GETs.

SET PROCEDURE

SET PROCEDURE instructs the compiler to locate the specified source code file and incorporate the contents into the current object file being created.

Deficiency: SET PROCEDURE is still around as a compatibility function. It serves no purpose in Clipper other than to obscure the structure and relationships of source code files. You must compile without the **/M** switch for a SET PROCEDURE to be effective. The rationale for using SET PROCEDURE is related to product design flaws never present in Clipper in the first place.

Remedy: If you want a particular object file to be included in your application you should make an explicit reference to it when linking. Or, if you want several source code files to be combined into a single object file you can create a .CLP file and compile it, instead.

SET UNIQUE

SET UNIQUE determines whether non-unique keys are included in index files.

Deficiency: SET UNIQUE is an application-wide setting and may introduce unintended side effects.

Remedy: Rather than using SET UNIQUE, take advantage of the INDEX command's UNIQUE clause so the creation of a unique index is limited to a single line, rather than the entire application. This eliminates toggling SET UNIQUE on and off all over your application and the resulting potential for mistakes.

```
index on Name to temp1
index on Zip to temp2 unique
```

STORE

STORE assigns a value to one or more variables.

Deficiency: Aside from being longer to type than the equivalent assignment operator, :=, using STORE brands you as a hopelessly old-fashioned dBASE bumbler.

Remedy: Use the assignment operators, := or =, wherever a STORE is encountered. The := operator is preferable to =, which should be reserved for use in logical comparisons.

```
store 0 to n    //  Your colleagues will snicker.
t := 0          //  Much better.
```

Prior to Clipper 5, programmers were required to use STORE to efficiently assign the same value to multiple variables. There is no harm in leaving existing code this way because the STORE command is preprocessed into a series of := operators, anyway. Converting a STORE command is easy. When writing new code you should get in the habit of using the := operator, just in case it becomes fashionable to laugh at even the previously legitimate use of the STORE command.

```
store 0 to a, b, c, d, e, f    //  These are equivalent.
a := b:= c := d := e := f := 0
```

TEXT..ENDTEXT

TEXT..ENDTEXT delimits a block of text to display.

Deficiency: Similar to DIR, TEXT..ENDTEXT is a chain saw in the toolbox of Clipper display capabilities. It offers almost no control over formatting and has no place in modern user interface design.

Remedy: There are numerous alternatives to TEXT..ENDTEXT, like using @..SAY or even storing the text to be displayed in a long string and displaying it via MEMOLINE(). Almost any solution is preferable to the unwieldy TEXT..ENDTEXT block.

With that said, there are indeed situations where it is useful to just dump a large amount of text to the screen or printer, but perhaps the question is not "should I use TEXT..ENDTEXT" but rather, "why is all this static text sitting in my source code in the first place?" What appears between TEXT and ENDTEXT is actually data, and good application design dictates that it be treated as such.

WAIT

WAIT pauses program execution and displays an optional message, keeps waiting until a key is pressed, and optionally stores a keypress to a variable.

Deficiency: WAIT is in the same boat as DIR and TEXT..ENDTEXT with respect to modern user interfaces. It isn't very pretty and has some major limitations on how it reacts to Ctrl and Alt keys (all return CHR(0)). Function keys are ignored unless they are assigned with a SET FUNCTION or SET KEY command. WAIT advances down one screen row regardless of whether a message is specified or not.

Remedy: A simple INKEY(0) is equivalent to WAIT, without any limitations on keystrokes. and it doesn't disturb the screen. You can store the result of INKEY(0) just as easily as with WAIT TO and gain access to the entire keyboard rather than the limited set of keys that WAIT will recognize. You can use the ?, ??, and @..SAY commands to display messages with more accuracy and flexibility.

```
wait "Your choice?" to c
                                        // These two are equivalent
? "Your choice?"
c := chr(inkey(0))
```

Obsolete functions

Ah, user-defined functions. Pioneered by Nantucket and even now not implemented as completely or successfully in other DBASE dialects, user-defined functions make it unlikely that you'll suffer very long should Nantucket drop a function you need. The functions currently considered obsolete will not be missed by most Clipper programmers. New programmers should not use them at all.

ADIR()

The ADIR() function fills a set of arrays with DOS directory information.

Deficiency: Saddled with a two-pass calling procedure and cursed with the need for up to five individual arrays, ADIR() is a needlessly clumsy way to get directory information.

Remedy: The DIRECTORY() function does everything ADIR() does, without requiring more than a single array.

AFIELDS()

The AFIELDS() function fills a series of arrays with database structure information.

Deficiency: Similar to ADIR(), AFIELDS() demands that a set of predefined arrays be available to receive various pieces of a database structure. Up to four individual arrays are required.

Remedy: The DBSTRUCT() function does everything AFIELDS() does, and needs only a single array to do it.

DBEDIT()

DBEDIT() displays database records in a table layout.

Deficiency: Don't even get us started — DBEDIT() is so pathetic in comparison to the TBrowse object class that it's not worth the time it takes to describe all the DBEDIT() deficiencies. Let's just say that DBEDIT() was a feeble attempt to answer the question, "Why can't Clipper support the BROWSE command?" Contrary to what the name DBEDIT() implies, it can't edit anything without loads of additional programming. Even the act of simply browsing through a database is an exercise in source code gymnastics.

Remedy: Gratefully abandon DBEDIT(), or don't get started in the first place. It isn't trivial but you can, without much effort, convert any DBEDIT() into a TBrowse. Along the way you'll discover how easy it is to implement fancy browsing features. Even TBrowse doesn't offer edit capabilities without some additional programming, but compared to DBEDIT() it's a walk in the park. See Chapter 25, "The TBrowse Object Class," for details.

DBF()

Returns name of current alias.

Deficiency: There's nothing inherently bad about DBF(), it's just superseded by the more flexible ALIAS() function.

Remedy: Use the ALIAS() function instead. ALIAS() is preferable because you can specify a work area while DBF() only operates on the current work area.

FKLABEL()

FKLABEL() returns the "name" of a function key.

Deficiency: FKLABEL()'s sole purpose is to make a very small percentage of DBASE applications compile under Clipper. All it does is stick an "F" on the front of the number you pass to it.

Remedy: If you actually managed to find a use for FKLABEL() we'd like to hear about it.

FKMAX()
FKMAX() returns the number of function keys present.

Deficiency: In Clipper this value is hard-wired at 40. Precious little is gained by that. Using 40 implies that there are 10 function keys and you can press them alone or in combination with the Shift, Ctrl and Alt keys.

Remedy: This is one function that could find some utility in Clipper if it returned the number of function keys actually present on the keyboard, allowing you to sense the F11 and F12 keys or who knows what else IBM will jam into the keyboards of the future. As it stands now you should disregard FKMAX() entirely; it's as close to a useless function as you're likely to encounter in Clipper.

MOD()
MOD() returns the modulus of two numbers and attempts to repeat some of the mistakes made in the dBASE III Plus implementation.

Deficiency: MOD() does mathematically incorrect things when presented with a zero divisor or negative numbers.

Remedy: Use Clipper's modulus operator, %, instead, and review all source code where MOD() is relied upon for correct results. (See Chapter 5, "Operators", for a detailed discussion of the modules operator and a comparison of "%", Clipper MOD(), and dBASE III MOD().)

READKEY()
READKEY() returns an indication of which key was used to exit a READ.

Deficiency: READKEY() emulates the function as it is implemented in dBASE III Plus. For whatever reason, it is designed to take what should be a simple READ exit status situation and make it incompatible with other keyboard events.

Remedy: Use a combination of LASTKEY() and UPDATED() to get the same information that's returned by READKEY(), but in a more standard manner that is compatible with other keyboard events.

RECCOUNT()

RECCOUNT() returns the number of records in a database file.

Deficiency: RECCOUNT() is identical in all respects to LASTREC(). Why we ever needed two functions for the same thing escapes logic, and evidently someone within Nantucket decided that the intuitive name RECCOUNT() had to go in favor of the more ambiguous LASTREC().

Remedy: RECCOUNT() can be replaced by LASTREC() in all situations with no side effects.

WORD()

WORD() converts numerics from type double to type int, but only in the context of a CALL command.

Deficiency: Because the CALL command is slated for obsolescence, it's not surprising that any functions related to it are also headed for the bit bucket.

Remedy: Convert routines currently used with the CALL command to use with the Clipper Extend System. Once the CALL is gone all occurrences of WORD() will be gone as well.

The End?

There will no doubt be more obsolete commands and functions in future releases of Clipper, as Nantucket attempts to wean us from our dBASE heritage and moves us into more efficient and more reliable programming techniques. If you eliminate the current list from your applications you stand a better chance of weathering future changes. Even if they weren't considered obsolete, many of the commands and functions aren't very well designed or implemented in the first place and you'll lose nothing by changing your programming habits.

Appendix A

STANDARD INKEY() KEY-CODE DEFINITIONS

```
// Cursor movement keys
#define K_UP             5
#define K_DOWN          24
#define K_LEFT          19
#define K_RIGHT          4
#define K_HOME           1
#define K_END            6
#define K_PGUP          18
#define K_PGDN           3
#define K_CTRL_LEFT     26
#define K_CTRL_RIGHT     2
#define K_CTRL_HOME     29
#define K_CTRL_END      23
#define K_CTRL_PGUP     31
#define K_CTRL_PGDN     30
#define K_CTRL_RET      10

#define K_ESC           27
#define K_RETURN        13
#define K_ENTER         13

// Editing keys
#define K_INS           22
#define K_DEL            7
#define K_BS             8
#define K_CTRL_BS      127
#define K_TAB            9
#define K_SH_TAB       271

// Control keys
#define K_CTRL_A         1
#define K_CTRL_B         2
#define K_CTRL_C         3
#define K_CTRL_D         4
#define K_CTRL_E         5
#define K_CTRL_F         6
#define K_CTRL_G         7
#define K_CTRL_H         8
#define K_CTRL_I         9
#define K_CTRL_J        10
#define K_CTRL_K        11
#define K_CTRL_L        12
#define K_CTRL_M        13
#define K_CTRL_N        14
#define K_CTRL_O        15
#define K_CTRL_P        16
#define K_CTRL_Q        17
#define K_CTRL_R        18
#define K_CTRL_S *      19
#define K_CTRL_T        20
#define K_CTRL_U        21
#define K_CTRL_V        22
#define K_CTRL_W        23
#define K_CTRL_X        24
#define K_CTRL_Y        25
#define K_CTRL_Z        26
```

```
// Alt keys
#define K_ALT_A         286
#define K_ALT_B         304
#define K_ALT_C       * 302
#define K_ALT_D       * 288
#define K_ALT_E         274
#define K_ALT_F         289
#define K_ALT_G         290
#define K_ALT_H         291
#define K_ALT_I         279
#define K_ALT_J         292
#define K_ALT_K         293
#define K_ALT_L         294
#define K_ALT_M         306
#define K_ALT_N         305
#define K_ALT_O         280
#define K_ALT_P         281
#define K_ALT_Q         272
#define K_ALT_R         275
#define K_ALT_S         287
#define K_ALT_T         276
#define K_ALT_U         278
#define K_ALT_V         303
#define K_ALT_W         273
#define K_ALT_X         301
#define K_ALT_Y         277
#define K_ALT_Z         300

// Function keys
#define K_F1             28
#define K_F2             -1
#define K_F3             -2
#define K_F4             -3
#define K_F5             -4
#define K_F6             -5
#define K_F7             -6
#define K_F8             -7
#define K_F9             -8
#define K_F10            -9
```

```
// Control-function keys
#define K_CTRL_F1       -20
#define K_CTRL_F2       -21
#define K_CTRL_F3       -22
#define K_CTRL_F4       -23
#define K_CTRL_F5       -24
#define K_CTRL_F6       -25
#define K_CTRL_F7       -26
#define K_CTRL_F8       -27
#define K_CTRL_F9       -28
#define K_CTRL_F10      -29

// Alt-function keys
#define K_ALT_F1        -30
#define K_ALT_F2        -31
#define K_ALT_F3        -32
#define K_ALT_F4        -33
#define K_ALT_F5        -34
#define K_ALT_F6        -35
#define K_ALT_F7        -36
#define K_ALT_F8        -37
#define K_ALT_F9        -38
#define K_ALT_F10       -39

// Shift-function keys
#define K_SH_F1         -10
#define K_SH_F2         -11
#define K_SH_F3         -12
#define K_SH_F4         -13
#define K_SH_F5         -14
#define K_SH_F6         -15
#define K_SH_F7         -16
#define K_SH_F8         -17
#define K_SH_F9         -18
#define K_SH_F10        -19
```

* ***WARNING: Before You Turn CTRL-S, ALT-C, or ALT-D Into Hot Keys***

These keys have previously defined meaning to a Clipper application:

CTRL-S pauses execution and removes key from buffer unless SCROLLBREAK is set off

ALT-C interrupts execution

ALT-D invokes debugger if present; otherwise it is ignored

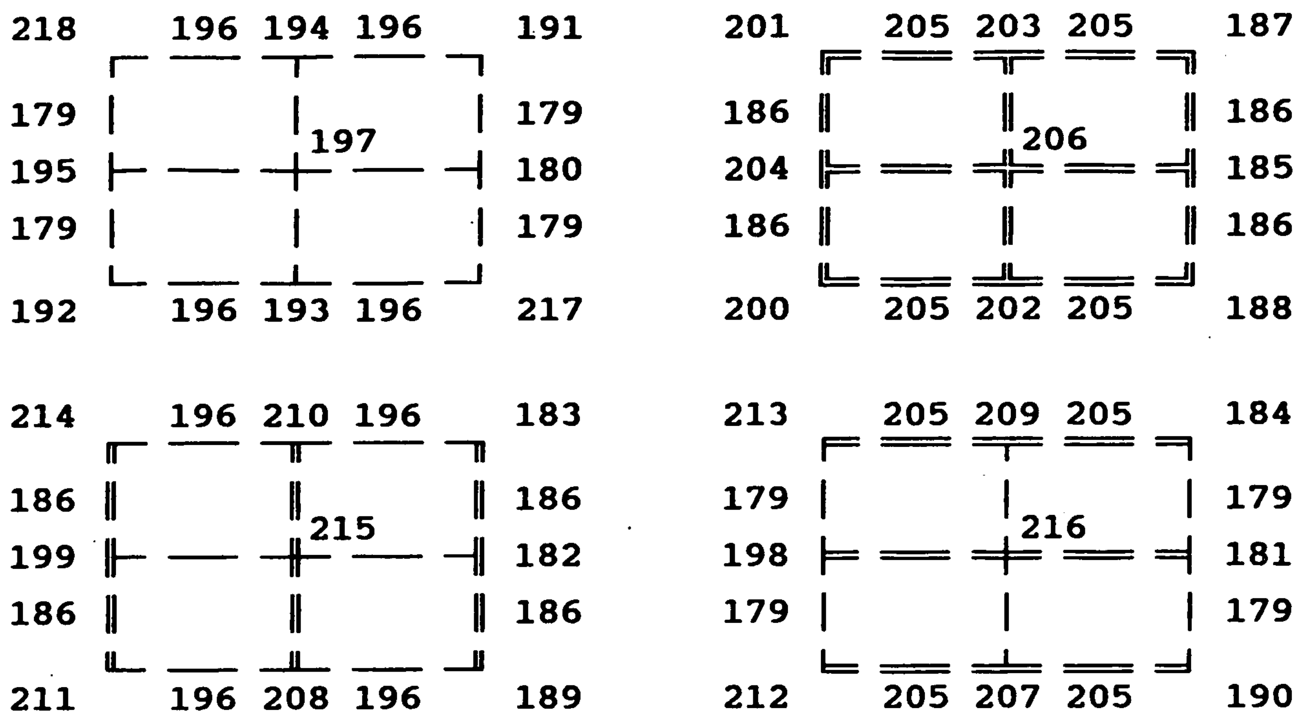
While it is possible to change the definition of these keys, it is best not to do so unless you must, and then be wary of potential side effects.

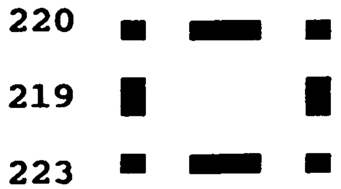
THE "IRRATIONAL" ASCII CHART

Val.	Char.	Val.	Char.	Val.	Char.	Val.	Char.	Val.	Char.
0		26		52	4	78	N	104	h
1	☺	27	←	53	5	79	O	105	i
2	☻	28	∟	54	6	80	P	106	j
3	♥	29	↔	55	7	81	Q	107	k
4	♦	30	▲	56	8	82	R	108	l
5	♣	31	▼	57	9	83	S	109	m
6	♠	32		58	:	84	T	110	n
7	•	33	!	59	;	85	U	111	o
8	◘	34	"	60	<	86	V	112	p
9	○	35	#	61	=	87	W	113	q
10		36	$	62	>	88	X	114	r
11	♂	37	%	63	?	89	Y	115	s
12	♀	38	&	64	@	90	Z	116	t
13	♪	39	'	65	A	91	[	117	u
14	♫	40	(	66	B	92	\	118	v
15	☼	41	)	67	C	93	]	119	w
16	►	42	*	68	D	94	^	120	x
17	◄	43	+	69	E	95	_	121	y
18	↕	44	,	70	F	96	`	122	z
19	‼	45	-	71	G	97	a	123	{
20	¶	46	.	72	H	98	b	124	¦
21	§	47	/	73	I	99	c	125	}
22	▬	48	0	74	J	100	d	126	~
23	↨	49	1	75	K	101	e	127	⌂
24	↑	50	2	76	L	102	f		
25	↓	51	3	77	M	103	g		

Val.	Char.	Val.	Char.	Val.	Char.	Val.	Char.	Val.	Char.
128	Ç	154	Ü	180	┤	206	╬	232	Φ
129	ü	155	¢	181	╡	207	╧	233	Θ
130	é	156	£	182	╢	208	╨	234	Ω
131	â	157	¥	183	╖	209	╤	235	δ
132	ä	158	₧	184	╕	210	╥	236	∞
133	à	159	ƒ	185	╣	211	╙	237	φ
134	å	160	á	186	║	212	╘	238	∈
135	ç	161	í	187	╗	213	╒	239	∩
136	ê	162	ó	188	╝	214	╓	240	
137	ë	163	ú	189	╜	215	╫	241	±
138	è	164	ñ	190	╛	216	╪	242	≥
139	ï	165	Ñ	191	┐	217	┘	243	≤
140	î	166	ª	192	└	218	┌	244	⌠
141	ì	167	º	193	┴	219	█	245	⌡
142	Ä	168	¿	194	┬	220	▄	246	÷
143	Å	169	⌐	195	├	221	▌	247	≈
144	É	170	¬	196	─	222	▐	248	°
145	æ	171	½	197	┼	223	▀	249	•
146	Æ	172	¼	198	╞	224	α	250	·
147	ô	173	¡	199	╟	225	β	251	√
148	ö	174	«	200	╚	226	Γ	252	n
149	ò	175	»	201	╔	227	π	253	2
150	û	176	░	202	╩	228	Σ	254	■
151	ù	177	▒	203	╦	229	σ	255	
152	ij	178	█	204	╠	230	µ		
153	Ö	179	│	205	═	231	τ		

ASCII CHART





254	■	176	░	79	O	224	α	239	∩
219	█	177	▒	111	o	225	ß	240	≡
220	▄	178	▓	248	°	226	Γ	241	±
221	▌	219	█	249	•	227	π	242	≥
222	▐			46	.	228	Σ	243	≤
223	▀			250	·	229	σ	244	⌠
						230	µ	245	⌡

131	â	146	Æ	139	ï	153	Ö	156	£	168	¿	231	τ	246	÷
132	ä	135	ç	140	î	150	û	157	¥	173	¡	232	Φ	247	≈
133	à	128	Ç	141	ì	151	ù	158	₧	169	⌐	233	Θ	248	°
134	å	130	é	161	í	163	ú	159	ƒ	170	¬	234	Ω	249	•
160	á	136	ê	162	ó	129	ü	164	ñ	171	½	235	δ	250	·
142	Ä	137	ë	147	ô	154	Ü	165	Ñ	172	¼	236	∞	251	√
143	Å	138	è	148	ö	152	ÿ	166	ª	174	«	237	φ	252	ⁿ
145	æ	144	É	149	ò	155	¢	167	º	175	»	238	ε	253	²

Appendix B

LONG SOURCE CODE LISTINGS

```
/*
    COLORS.CH   Manifest Constants for Color Management System
*/

#define  C_NORMAL     1
#define  C_BOLD       2
#define  C_ENHANCED   3
#define  C_BLINK      4
#define  C_MESSAGE    5
#define  C_WARNING    6
/* eof Colors.Ch */


/*
   MSGBOX.CH   Preprocessor definition for MsgBox(), a user
    interface building block function.
*/
#translate MESSAGE <msg,...> ;
             [ AT <r1> [, <c1>] ] ;
             [ TO <r2> [, <c2>] ] ;
             [ WIDTH <w> ] ;
             [ DEPTH <d> ] ;
             [ COLOR <clr> ] ;
             [ CHOOSE <ch,...> ] ;
             [ CHOOSECOLOR <chClr> ] ;
             [ CHCOLOR <chClr> ] ;
             [ INTO <ret> ] ;
             [ WAIT <wait> ] ;
             [ <rest:RESTORE,REST> ] ;
  => ;
        [<ret> := ] MsgBox( <r1>, <c1>, <r2>, <c2>, <w>, <d>, ;
                            {<msg>}, <clr>, {<ch>}, <chClr>, ;
                            <wait>, <.rest.> )
/* eof MsgBox.Ch */
```

(19)日本国特許庁（ＪＰ）　(12) **公開特許公報**（Ａ）　(11)特許出願公開番号
特開平5－27947
(43)公開日　平成5年(1993)2月5日

(51)Int.Cl.⁵	識別記号	庁内整理番号	ＦＩ	技術表示箇所
Ｇ０６Ｆ　7/24		8323－5B		
15/02	３３０　Ｐ	9194－5L		

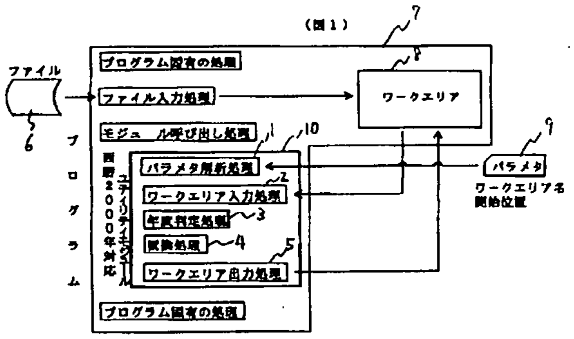
審査請求　未請求　請求項の数1（全　4　頁）

(21)出願番号	特願平3－177982
(22)出願日	平成3年(1991)7月18日
(71)出願人	000005108 株式会社日立製作所 東京都千代田区神田駿河台四丁目6番地
(72)発明者	狭間　正和 神奈川県川崎市幸区鹿島田890番地の12 株式会社日立製作所情報システム開発本部内
(74)代理人	弁理士　小川　勝男

(54)【発明の名称】　年度順保証方法

(57)【要約】
【目的】データファイル中の西暦の下２ケタで年度を管理している場合でも、現状のデータファイルフォーマットで、西暦２０００年以降も年度順を保証する。
【構成】年度の大小判定，ソート・マージ処理の前に、西暦２０００年対応ユティリティモジュール１０を起動する。その後、外部パラメタ９により予め西暦下２ケタが格納されているレコード中の位置を指定し、予め指定した範囲の西暦を示すコードを、年度の順序が維持される様に他のコードを置き換える。

（図１）



【特許請求の範囲】
【請求項１】記憶手段と処理部を有するコンピュータシステムにおいて、前記記憶手段に１９００年代と２０００年代の西暦の下２桁が格納されている場合、処理部は、西暦の下２桁のうち十の位のコードを、年度の順序を維持するコードに置き換えることを特徴とする年度順保証方法。
【発明の詳細な説明】
【０００１】
【産業上の利用分野】本発明は、西暦の下２桁が格納されているデータファイルに対する、プログラムによる大小判定及びソート・マージ処理での昇降順を保証し、正常な結果を得る西暦２０００年に対応した年度順保証方法に関するものである。
【０００２】
【従来の技術】従来のコンピュータシステムでは、データファイル中に、西暦の下２桁が格納されている。なお、この種の技術に関するものとしては、例えば、特開昭５８－１２２９号公報や、特開平３－２２１１７号公報がある。
【０００３】
【発明が解決しようとする課題】上記従来技術は、西暦２０００年以降の場合を考慮しておらず、データファイル中の西暦の下２桁で年度を管理している。そのため、西暦２０００年以降では、通常の年度による大小判定処理及びソート・マージ処理での昇降順序処理を行った場合、その大小関係は、数１で表わされる。
【０００４】
【数１】

$$19\underline{99} > 19\underline{98} > 20\underline{01} > 20\underline{00} \quad \cdots（数１）$$

【０００５】つまり、２０００年は、１９９９年よりも大きいと判断されなければならないにも関わらず、判断にあたっては、西暦の下２桁しか用いないので、００は９９より小さいことから、２０００年は１９９９年よりも小さいと判断されてしまう。これを解決する方法として、西暦を４桁にすることが考えられるが、この場合、データファイルのレコード長，ブロック長を変更することが必要となり、また、プログラム修正も発生してしまうという問題点がある。
【０００６】本発明の目的は、コードの体系を生かし、現状のデータファイルフォーマットで西暦２０００年以降も対応可能な西暦２０００年に対応した年度順保証方法を提供することにある。
【０００７】
【課題を解決するための手段】上記目的を達成するために、記憶手段と処理部を有するコンピュータシステムにおいて、前記記憶手段に１９００年代と２０００年代の西暦の下２桁が格納されている場合、処理部は、西暦の下２桁のうち十の位のコードを、年度の順序を維持するコードに置き換えることにしたものである。
【０００８】
【作用】１９００年代と２０００年代の西暦を示すデータが混在する場合、年度の順序が維持される様に、西暦を示すデータのコードを他のコードと置き換える。これにより、大小判定、昇降順序処理が保証される。
【０００９】
【実施例】図１は、本発明の一実施例のプログラムの構成図である。
【００１０】６は、下２桁のみの西暦を含むデータが格納されたファイルである。７は、プログラムである。８は、ワークエリアである。９は、パラメタである。１０は、西暦２０００年対応ユティリティモジュールである。
【００１１】西暦２０００年対応ユティリティモジュール１０（以下モジュール１０とする）は、プログラム７やユティリティで指定されると動作し、年度を取扱う処理の前処理として位置付けられる。なお、モジュール１０には、予め、コード変換を行なう下２桁の範囲を指定しておく。置き換えるのは、１９００年代で下２桁が一番小さい数よりも下２桁が小さい２０００年代の数である。例えば、ファイル６中のデータの西暦が１９７３年から始まる場合は、下２桁が００（２０００年）から７２（２０７２年）までを変換する。本実施例は、ファイル６には１９６０年からのデータが有る場合の例でなので、下２桁が００～５９の場合にコード変換をするように範囲を指定する。なお、本実施例では、コードとしてＥＢＣＤＩＣコードを用いている。
【００１２】次に、処理手順を説明する。まず、プログラム７は、年度判定を行う前段階でモジュール１０の呼び出し処理を行う。その後、モジュール１０では以下の処理を行う。
【００１３】パラメタ解析処理１：外部より与えられたパラメタ９を入力して内容を解析し、ワークエリア名及びレコード中の西暦下２桁の開始位置を認識する。
【００１４】ワークエリア入力処理２：パラメタ解析処理１で得たワークエリア名からデータを入力する。
【００１５】年度判定処理３：パラメタ解析処理１で得た、レコード中の西暦下２桁の開始位置からの２バイトを判定し、一定の範囲内の時、本実施例では’００’から’５９’の範囲の場合に置換処理４を行う。それ以外は、次のデータを入力する。
【００１６】置換処理４：レコード中の西暦下２桁のうち十の位を表１の様に置換する。
【００１７】

【表1】

（表１）

	置換前	置換後
十の位が０の場合	X’ F０’	X’ FA’
十の位が１の場合	X’ F１’	X’ FB’
十の位が２の場合	X’ F２’	X’ FC’
十の位が３の場合	X’ F３’	X’ FD’
十の位が４の場合	X’ F４’	X’ FE’
十の位が５の場合	X’ F５’	X’ FF’

【００１８】ワークエリア出力処理５：置換処理４を行ったデータをワークエリア８に出力する。

【００１９】表１に示される様に、文字コードを置き換えることにより、１９９０年よりも２０００年が、２０００年代より２０１０年が大きいと判断される。

【００２０】なお、本実施例では、ＥＢＣＤＩＫコードで、２０００年以上の場合、表１に示すコードの置き換え処理を行ったが、２０００年未満の場合に、表２の様に、空いているコードと置き換えてもよい。このやり方では、例えば、ファイル６に１９９９年からのデータが有る場合ならば、２０９８年まで対応できる。

【００２１】

【表2】

（表２）

	置換前	置換後
十の位が０の場合	X’ F０’	X’ B０’
十の位が１の場合	X’ F１’	X’ B１’
十の位が２の場合	X’ F２’	X’ B２’
十の位が３の場合	X’ F３’	X’ B３’
十の位が４の場合	X’ F４’	X’ B４’
十の位が５の場合	X’ F５’	X’ B５’
十の位が６の場合	X’ F６’	X’ B６’
十の位が７の場合	X’ F７’	X’ B７’
十の位が８の場合	X’ F８’	X’ B８’
十の位が９の場合	X’ F９’	X’ B９’

【００２２】また、２０００年未満の場合に、X’ F０’→X’ C０’等、使用しているコードと置き換えてもよい。

【００２３】また、２０００年代の場合に、X’ F０’→X’ C０’と、１９００年代の場合に、X’ F０’→X’ B０’と置き換えてもよい。つまり、２０００年代と１９００年代の両方を他のコードに置き換えてもよい。

【００２４】また、使用する文字コードはＪＩＳコードやＡＳＣＩＩコード等でも構わない。　つまり、年度の大小関係等を判断するときに、判断が正常に行える様にコードの置き換えを行えばよい。

【００２５】

【発明の効果】年度の大小関係判定が正常に行える様にコードの置き換えを行うことにより、データファイルのレコード長，ブロック長を変更せず、またプログラム修正もせずに年度の順序が保証されるという効果がある。

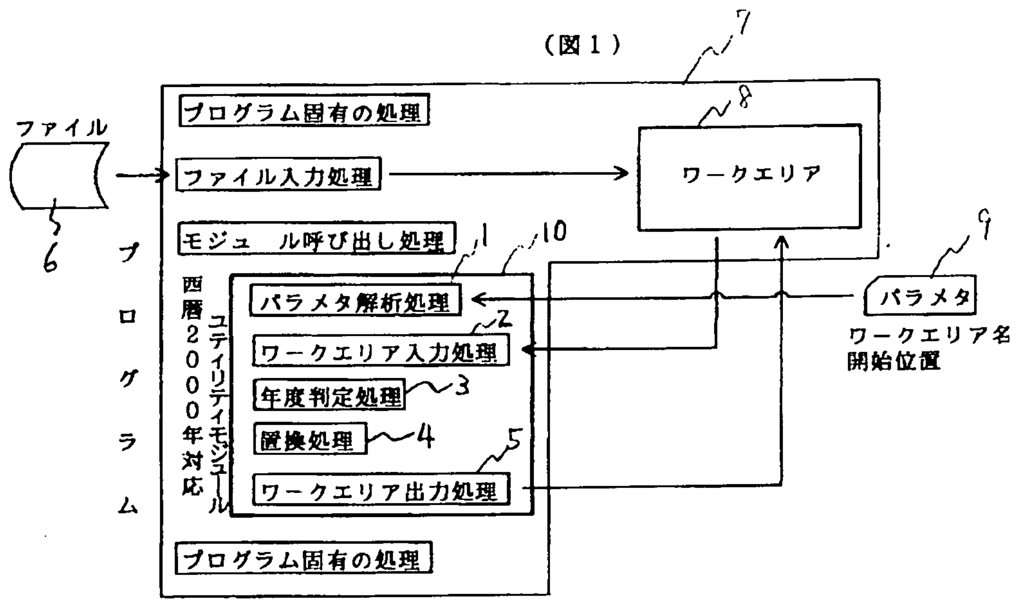
【００２６】

【図面の簡単な説明】

【図１】プログラムの構成を示す図である。

【符号の説明】
1…パラメタ解析処理、2…ワークエリア入力処理、3…年度判定処理、4…置換処理、5…ワークエリア出力処理、6…ファイル、7…プログラム、8…ワークエリア、9…パラメタ、10…西暦２０００年対応ユティリティモジュール。

【図１】



OIPE JC16 NOV 0 9 1999 PATENT & TRADEMARK OFFICE

5,806,063 #16

Under the authority of 37 CFR 1.501 and the patentability requirements of 35 USC 102(a) and (b) I wish to cite prior art relating to U.S. patent number 5,806,063, "Date formatting and sorting for dates spanning the turn of the century" issued on September 8, 1998 to Bruce Dickens.

I wish to bring to the attention of the Patent Office the below cited prior art which I believe bears on the claims of the patent. Pursuant to 35 USC 102(a) and (b), some of the methods claimed in the patent were known and used by others and described in printed publications in this and foreign countries more than one year prior to the date of the application for patent. In particular, the method of claim 1, where dates are represented symbolically with a two-digit year and interpreted with respect to a "10-decade window", has been used for at least 15 years. The method is mandated in the ANSI Common Lisp standard[1], as described in section 25.1.4.1, "Decoded Time", of that standard. That section has a long history; the technique is also described in *Common Lisp, The Language, Second Edition.*[2] (It is also described in the first edition of that work.[3]) The same method is described differently in the *Lisp Machine Manual*, chapter 36, "Dates and Times".[4] The application to interpretation of dates stored in databases is immediate — indeed one would have to go out of their way to avoid it if the database application were written in Common Lisp. Further, at least two database vendors have, at least four years prior to the filing of this patent application, defined this method for interpreting two-digit year representations.[5,6]

Secondly, with respect to dependent claim 5 and the final step of claim 1 (which, not incidentally, appear identical), the described date format is previously described in International Standard ISO 8601 : 1988 (E)[7], and also in the earlier ISO 2014-1976 (E)[8], which the former supercedes. The usefulness of this date format for automated manipulation is expressly mentioned in the ISO standards; the application to sorting (dependent claims 4 and 6 of the patent) is immediately obvious. The ISO standards are also described and referenced in Internet RFC 753[9], among others.

---

[1] ANSI X3.226-1994, "Information Technology — Programming Language — Common Lisp", 1994. Section 25.1.4.1.

[2] Guy Steel, *Common Lisp, The Language, Second Edition,* Digital Equipment Corporation, 1990, p.702.

[3] Guy Steel, *Common Lisp, The Language,* Digital Equipment Corporation, 1984.

[4] David Moon, et. al., *Lisp Machine Manual,* Sixth Edition, MIT Artificial Intelligence Laboratory, 1984, p. 776. Note: the "Compatibility Note" in *Common Lisp, The Language* referring to Lisp Machine Lisp was referring to an earlier edition of that language, prior to the introduction of the rolling "10-decade" window. The fourth edition of the *Lisp Machine Manual,* for instance, does not include the feature.

[5] *Commands Reference Manual for Sybase SQL Server for UNIX,* Release 4.9.1,. Document ID 32270-01-0491-01, 1 October 1992. Chapter 2, Section "Date Functions", p 2-106.

[6] Oracle Corporation, *SQL Language Reference Manual,* Version 7.0, Part number 778-70-0292, May 1992. p. 4-49.

[7] *International Standard ISO 8601 : 1988 (E),* "Data elements and interchange formats — Information exchange — Representation of dates and times", International Organization for Standardization, first edition 1988-06-15.

[8] *International Standard ISO 2014 : 1976 (E),* "Writing of calendar dates in all-numeric form", International Organization for Standardization, first edition 1976-04-01.

[9] Jonathan Postel, Internet RFC 753, "Internet Message Protocol", March 1979.

Sheet 1 of 2

37 CFR 1.501 INFORMATION DISCLOSURE CITATION IN A PATENT (Use several sheets if necessary)	Docket Number (Optional)	Patent Number 5,806,063
	Applicant Bruce Dickens	
	Issue Date Sep 8, 1998	Group Art Unit

U. S. PATENT DOCUMENTS

EXAMINER INITIAL	DOCUMENT NUMBER	DATE	NAME	CLASS	SUBCLASS	FILING DATE IF APPROPRIATE

FOREIGN PATENT DOCUMENTS

	DOCUMENT NUMBER	DATE	COUNTRY	CLASS	SUBCLASS	Translation YES	Translation NO

OTHER DOCUMENTS (Including Author, Title, Date, Pertinent Pages, Etc.)

	ANSI X3.226-1994, Programming Language Common Lisp, 1994 Section 25.1.4.1
	Guy Steel, Common Lisp, The Language, Second Edition, 1990, p702
	Commands Reference Manual for SQL Server, Release 4.9.1 Sybase Corporation, October 1, 1992, Chapter 2, Section "Date Functions" p 2-106

EXAMINER	DATE CONSIDERED

PTO/SB/42 (10-96)
Approved for use through 10/31/99. OMB 0651-0031
Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

Sheet 2 of 2

37 CFR 1.501 INFORMATION DISCLOSURE CITATION IN A PATENT

(Use several sheets if necessary)

Docket Number (Optional)	Patent Number 5,806,063
Applicant Bruce Dickens	
Issue Date Sep 8, 1998	Group Art Unit

OIPE JC16 NOV 09 1999 PATENT & TRADEMARK OFFICE

U. S. PATENT DOCUMENTS

EXAMINER INITIAL	DOCUMENT NUMBER	DATE	NAME	CLASS	SUBCLASS	FILING DATE IF APPROPRIATE

FOREIGN PATENT DOCUMENTS

	DOCUMENT NUMBER	DATE	COUNTRY	CLASS	SUBCLASS	Translation YES	Translation NO

OTHER DOCUMENTS (Including Author, Title, Date, Pertinent Pages, Etc.)

Oracle Corporation, SQL Language Reference Manual, Version 7.0, May 1992, p. 4-49

International Standard ISO 8601:1988 (E), International Organization for Standardization, 1998-06-15

EXAMINER	DATE CONSIDERED

Burden Hour Statement: This form is estimated to take 2.0 hours to complete. Time will vary depending upon the needs of the individual case. Any comments on the amount of time you are required to complete this form should be sent to the Chief Information Officer, Patent and Trademark Office, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Assistant Commissioner for Patents, Washington, DC 20231.

25.1.4.1 Decoded Time

A *decoded time* is an ordered series of nine values that, taken together, represent a point in calendar time (ignoring *leap seconds*):

Second

An *integer* between 0 and 59, inclusive.

Minute

An *integer* between 0 and 59, inclusive.

Hour

An *integer* between 0 and 23, inclusive.

Date

An *integer* between 1 and 31, inclusive (the upper limit actually depends on the month and year, of course).

Month

An *integer* between 1 and 12, inclusive; 1 means January, 2 means February, and so on; 12 means December.

Year

An *integer* indicating the year A.D. However, if this *integer* is between 0 and 99, the ``obvious" year is used; more precisely, that year is assumed that is equal to the *integer* modulo 100 and within fifty years of the current year (inclusive backwards and exclusive forwards). Thus, in the year 1978, year 28 is 1928 but year 27 is 2027. (Functions that return time in this format always return a full year number.)

Day of week

An *integer* between 0 and 6, inclusive; 0 means Monday, 1 means Tuesday, and so on; 6 means Sunday.

Daylight saving time flag

A *generalized boolean* that, if *true*, indicates that daylight saving time is in effect.

Time zone

A *time zone*.

The next figure shows *defined names* relating to *decoded time*.

```
decode-universal-time  get-decoded-time
```

Figure 25-5. Defined names involving time in Decoded Time.

Starting points | Contents Chapter 1 Chapter 2 | Master Index | Symbol Index | Glossary, n. Index of terms. | x3j13 issues

COMMON LISP

THE LANGUAGE

SECOND EDITION

GUY L. STEELE JR.

with contributions by

SCOTT E. FAHLMAN
RICHARD P. GABRIEL
DAVID A. MOON
DANIEL L. WEINREB

and with contributions to the second edition by

DANIEL G. BOBROW
LINDA G. DEMICHIEL
RICHARD P. GABRIEL
SONYA E. KEENE
GREGOR KICZALES
DAVID A. MOON
CRISPIN PERDUE
KENT M. PITMAN
RICHARD C. WATERS
JON L WHITE

digital™
Digital Press

9 8 7 6 5 4

Printed in the United States of America.

Design: David Ford
Copyediting: Alice Cheyer and Kate Schmit
Index: Marilyn Rowland
Composition: Guy L. Steele Jr. (See Colophon.)
Camera-ready copy: Advanced Computer Graphics
Production: Editorial Inc.
Printer: Hamilton Printing Company

Library of Congress Cataloging-in-Publication Data

Steele, Guy.
COMMON LISP.

Includes bibliography references (p.)
1. COMMON LISP (Computer program language) I. Title.
QA76.73.L23S73 1990 005.13'3 89-26016

Paperback order number EY-C187E-DP ISBN 1-55558-041-6

Contents

25.4.1. Time Functions

Time is represented in three different ways in Common Lisp: Decoded Time, Universal Time, and Internal Time. The first two representations are used primarily to represent calendar time and are precise only to one second. Internal Time is used primarily to represent measurements of computer time (such as run time) and is precise to some implementation-dependent fraction of a second, as specified by `internal-time-units-per-second`. Decoded Time format is used only for absolute time indications. Universal Time and Internal Time formats are used for both absolute and relative times.

Decoded Time format represents calendar time as a number of components:

- *Second*: an integer between 0 and 59, inclusive.
- *Minute*: an integer between 0 and 59, inclusive.
- *Hour*: an integer between 0 and 23, inclusive.
- *Date*: an integer between 1 and 31, inclusive (the upper limit actually depends on the month and year, of course).
- *Month*: an integer between 1 and 12, inclusive; 1 means January, 12 means December.
- *Year*: an integer indicating the year A.D. However, if this integer is between 0 and 99, the "obvious" year is used; more precisely, that year is assumed that is equal to the integer modulo 100 and within fifty years of the current year (inclusive backwards and exclusive forwards). Thus, in the year 1978, year 28 is 1928 but year 27 is 2027. (Functions that return time in this format always return a *full* year number.)

Compatibility note: This is incompatible with the Lisp Machine Lisp definition in two ways. First, in Lisp Machine Lisp a year between 0 and 99 always has 1900 added to it. Second, in Lisp Machine Lisp time functions return the abbreviated year number between 0 and 99 rather than the full year number. The incompatibility is prompted by the imminent arrival of the twenty-first century. Note that `(mod` *year* `100)` always reliably converts a year number to the abbreviated form, while the inverse conversion can be very difficult.

- *Day-of-week*: an integer between 0 and 6, inclusive; 0 means Monday, 1 means Tuesday, and so on; 6 means Sunday.
- *Daylight-saving-time-p*: a flag that, if not `nil`, indicates that daylight saving time is in effect.

- *Time-zone*: an integer specified as the numbe
Mean Time). For example, in Massachusetts
it is 8. Any adjustment for daylight saving

X3J13 voted in March 1989 ⟨178⟩ to specif
Time need not be an integer, but may be any
a ratio) in the range -24 to 24 (inclusive on bo
of 1/3600.

Rationale: For all possible time designations to be
the time zone to be non-integral, for some places
from Greenwich Mean Time by a non-integral num
There appears to be no user demand for floati
would introduce inexact arithmetic. X3J13 did not
This specification does require time zones to b
second (rather than 1 hour). This prevents problems
Decoded Time to Universal Time.

Universal Time represents time as a singl
time purposes, this is a number of seconds.
of seconds since midnight, January 1, 1900
(that is, 12:00:01 A.M.) on January 1, 1900 C
corresponds to time 00:00:01 on January 1.
was *not* a leap year; for the purposes of Com
only if its number is divisible by 4, except th
years, except that years divisible by 400 *are*
will be a leap year. (Note that the "leap seco
the world's official timekeepers as an additi
Lisp assumes that every day is exactly 86400
is used as a standard time representation with
Because the Common Lisp Universal Time
integers, times before the base time of midn
processed by Common Lisp.

Internal Time also represents time as a
implementation-dependent unit. Relative ti
units. Absolute time is relative to an arbi
which the system began running.

`get-decoded-time`

The current time is returned in Decoded Ti

Commands Reference Manual for SYBASE SQL Server™ for UNIX

SYBASE SQL Server Release 4.9.1

Document ID: 32270-01-0491-01

Change Level: 2

Last Revised: October 1, 1992

Date Functions

Function

Used to manipulate *datetime* values.

Syntax

All date functions take arguments except getdate. Function names, arguments, and results are listed in the following table.

Function	Argument	Result
getdate	()	Returns the current system date and time.
datename	(*datepart, date*)	Returns the name of the specified part (such as the month "June") of a *datetime* value, as a character string. If the result is numeric, such as "23" for the day, it is still returned as a character string.
datepart	(*datepart, date*)	Returns an integer value for the specified part of a *datetime* value.
datediff	(*datepart, date1, date2*)	Returns *date2 - date1*, measured in the specified

datename – produces the specified datepart (the first argument) of the specified date (the second argument) as a character string. Takes either a *datetime* or *smalldatetime* value as its second argument.

datepart – produces the specified date part (the first argument) of the specified date (the second argument) as an integer. Takes either a *datetime* or *smalldatetime* value as its second argument.

It is also used as an argument with **dateadd, datediff, datename**, and **datepart**. The following table lists the date parts, the abbreviations recognized by SQL Server, and the acceptable values.

Date Part	Abbreviation	Values
year	yy	1753 - 9999 (2079 for *smalldatetime)*
quarter	qq	1 - 4
month	mm	1 - 12
dayofyear	dy	1 - 366
day	dd	1 - 31
week	wk	1 - 54
weekday	dw	1 - 7 (Sun.-Sat.)
hour	hh	0 - 23
minute	mi	0 - 59
second	ss	0 - 59
millisecond	ms	0 - 999

Table 2-13: Date Parts and Their Values

If the year is given with two digits, <50 is the next century ("25" is "2025") and >=50 is this century ("50" is "1950").

Milliseconds can be preceded either with a colon or a period. If preceded by a colon, the number means thousandths of a second. If preceded by a period, a single digit means tenths of a second, two digits mean hundredths of a second, and three digits mean thousandths of a second. For example, "12:30:20:1" means twenty and one-thousandth of a second past 12:30; "12:30:20.1" means twenty and one-tenth of a second past 12:30.

date – an argument used with **dateadd, datediff, datename**, and **datepart**. The date can be either the function **getdate**, a character string in one of the acceptable date formats (see "Datatypes"), an expression that evaluates to a valid date format, or the name of a *datetime* column.

datediff – calculates the number of date parts between two specified dates. It takes three arguments. The first is a date part. The second and third are dates, either *datetime* or *smalldatetime* values. The result is a signed integer value equal to *date2 - date1*, in dateparts.

SQL LANGUAGE REFERENCE MANUAL

VERSION 7.0
PART NUMBER 778-70-0292
DEVELOPER'S RELEASE DOCUMENTATION
MAY 1992

ORACLE®

The Relational Database Management System

SQL Language Reference Manual
Version 7.0

Part No. 778-70-0292
Developer's Release Documentation
May 1992

Contributing Author: Brian Linden

Contributors: Pending

Date Format Elements and National Language Support

The functionality of some date format elements depend upon the country and language in which you are using ORACLE. For example, these date format elements return spelled values:

- MONTH
- MON
- DAY
- DY
- BC or AD or B.C. or A.D.
- AM or PM or A.M. or P.M.

The language in which these values are returned is specified either explicitly with the initialization parameter NLS_DATE_LANGUAGE or implicitly with the initialization parameter NLS_LANGUAGE. The values returned by the YEAR and SYEAR date format elements are always in English.

The date format element D returns the number of the day of the week (1-7). The day of the week that is numbered 1 is specified implicitly by the initialization parameter NLS_TERRITORY.

For complete information on National Language Support, including these initialization parameters, see Appendix G "National Language Support" of the *ORACLE RDBMS Database Administrator's Guide*.

ISO Standard Date Format Elements

ORACLE calculates the values returned by the date format elements IYYY, IYY, IY, I, and IW according to the ISO standard. For information on the differences between these values and those returned by the date format elements YYYY, YYY, YY, Y, and WW, see Appendix G "National Language Support" of the *ORACLE RDBMS Database Administrator's Guide.*

The RR Date Format Element

The RR date format element is similar to the YY date format element, but it provides additional flexibility for storing date values in other centuries. The RR date format element allows you to store twenty-first century dates in the twentieth century by specifying only the last two digits of the year. It will also allow you to store twentieth century dates in the twenty-first century in the same way if necessary.

If you use the TO_DATE function with the YY date format element, the date value returned is always in the current century. If you use the RR date format element instead, the century of the return value varies according to the specified two-digit year and the last two digits of the current year. Table 4-14 summarizes the behavior of the RR date format element.

TABLE 4-14
The RR Date Format Element

		If the specified two-digit year is	
		0 - 49	50 - 99
If the last two digits of the current year are:	0 - 49	The return date is in the current century.	The return date is in the century before the current one.
	50 - 99	The return date is in the century after the current one.	The return date is in the current century.

The following example also demonstrate the behavior of the RR date format element.

Example IV Assume these queries are issued prior to the year 2000:

```
SELECT TO_CHAR(TO_DATE('27-OCT-95', 'DD-MON-RR') ,'YYYY') "4-digit year"
    FROM DUAL

4-digit year
------------
        1995


SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "4-digit year"
    FROM DUAL

4-digit year
------------
        2017
```

Assume these queries are issued in the year 2000 or after:

```
SELECT TO_CHAR(TO_DATE('27-OCT-95', 'DD-MON-RR') ,'YYYY') "4-digit year"
    FROM DUAL

4-digit year
------------
        1995


SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "4-digit year"
    FROM DUAL

4-digit year
------------
        2017
```

Note that the queries return the same values regardless of whether they are issued before or after the year 2000. The RR date format element allows you to write SQL statements that will return the same values after the turn of the century.

INTERNATIONAL STANDARD

ISO
8601
First edition
1988-06-15

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION
МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ

Data elements and interchange formats — Information interchange — Representation of dates and times

Éléments de données et formats d'échange — Échange d'information — Représentation de la date et de l'heure

Reference number
ISO 8601 : 1988 (E)

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council. They are approved in accordance with ISO procedures requiring at least 75 % approval by the member bodies voting.

International Standard ISO 8601 was prepared by Technical Committee ISO/TC 154, *Documents and data elements in administration, commerce and industry.*

It cancels and replaces International Standards ISO 2014 : 1976, ISO 2015 : 1976, ISO 2711 : 1973, ISO 3307 : 1975 and ISO 4031 : 1978, of which it constitutes a technical revision.

Users should note that all International Standards undergo revision from time to time and that any reference made herein to any other International Standard implies its latest edition, unless otherwise stated.

Contents

Data elements and interchange formats — Information interchange — Representation of dates and times

0 Introduction

0.1 Although ISO Recommendations and Standards in this field have been available since 1971, different forms of numeric representation of dates and times have been in common use in different countries. Where such representations are interchanged across national boundaries misinterpretation of the significance of the numerals can occur, resulting in confusion and other consequential errors or losses. The purpose of this International Standard is to eliminate the risk of misinterpretation and to avoid the confusion and its consequences.

0.2 This International Standard includes specifications for the numeric representation of information regarding date and time of the day.

0.3 In order to achieve similar formats for the representations of calendar dates, ordinal dates, dates identified by week number, periods of time, combined date and time of the day, and differences between local time and Coordinated Universal Time, and to avoid ambiguities between these representations, it has been necessary to use, apart from numeric characters, either single alphabetic characters or one or more other graphic characters or a combination of alphabetic and other characters in some of the representations.

0.4 The above action has had the benefit of enhancing the versatility and general applicability of previous International Standards in this field, and provides for the unique representation of any date or time expression or combination of these. Each representation can be easily recognized, which is beneficial when human interpretation is required.

0.5 This International Standard retains the most commonly used expressions for date and time of the day and their representations from the earlier International Standards and provides unique representations for some new expressions used in practice. Its application in information interchange, especially between data processing systems and associated equipment will eliminate errors arising from misinterpretation and the costs these generate. The promotion of this International Standard will not only facilitate interchange across international boundaries, but will also improve the portability of software, and will ease problems of communication within an organization, as well as between organizations.

0.6 Several of the alphabetic and graphic characters used in the text of this International Standard are common both to the representations specified and to normal typographical presentation.

0.7 To avoid confusion between the representations and the actual text, its punctuation marks and associated graphic characters, all the representations are contained in brackets [ ]. The brackets are not part of the representation, and should be omitted when implementing the representations. All matter outside the brackets is normal text, and not part of the representation. In the associated examples, the brackets and typographical markings are omitted.

1 Scope and field of application

This International Standard specifies the representation of dates in the Gregorian calendar and times and representations of periods of time. It includes

a) calendar dates expressed in terms of year, month and day of month;

b) ordinal dates expressed in terms of year and day of year;

c) dates identified by means of year, week numbers and day numbers;

d) time of the day based upon the 24-hour timekeeping system;

e) differences between local time and Coordinated Universal Time (UTC);

f) combination of date and time;

g) periods of time, with or without either a start or end date or both.

This International Standard is applicable whenever dates and times are included in information interchange.

This International Standard does not cover dates and times where words are used in the representation.

This International Standard does not assign any particular meaning or interpretation to any data element that uses representations in accordance with this International Standard. Such meaning will be determined by the context of the application.

2 References

ISO 31-0 : 1981, *General principles concerning quantities, units and symbols.*

ISO 31-1 : 1978, *Quantities and units of space and time.*

ISO 646 : 1983, *Information processing — ISO 7-bit coded character set for information interchange.*

3 Terms and definitions

For the purposes of this International Standard, the following terms and definitions apply.

3.1 complete representation: The representation that includes all the date and time elements associated with the expression.

3.2 Coordinated Universal Time (UTC): The time scale maintained by the Bureau International de l'Heure (International Time Bureau) that forms the basis of a coordinated dissemination of standard frequencies and time signals.

NOTES

1 The source of this definition is Recommendation 460-2 of the Consultative Committee on International Radio (CCIR). CCIR has also defined the acronym for Coordinated Universal Time as UTC (see also 5.3.3).

2 UTC is often (incorrectly) referred to as Greenwich Mean Time and appropriate time signals are regularly broadcast.

3.3 date, calendar: A particular day of a calendar year, identified by its ordinal number within a calendar month within that year.

3.4 date, ordinal: A particular day of a calendar year identified by its ordinal number within the year.

3.5 day: A period of time of 24 hours starting at 0000 and ending at 2400 (which is equal to the beginning of 0000 the next day).

3.6 format, basic: The format of a representation comprising the minimum number of components necessary for the precision required.

3.7 format, extended: An extension of the basic format that includes additional separators.

3.8 Gregorian calendar: A calendar in general use introduced in 1582 to correct an error in the Julian calendar. In the Gregorian calendar common years have 365 days and leap years 366 days divided into 12 sequential months.

3.9 hour: A period of time of 60 minutes.

3.10 local time: The clock time in public use locally.

3.11 minute: A period of time of 60 seconds.

3.12 month, calendar: A period of time resulting from the division of a calendar year in twelve sequential periods of time, each with a specific name and containing a specified number of days. In the Gregorian calendar, the months of the calendar year, listed in their order of occurrence, are named and contain the number of days as follows: January (31), February (28 in common years; 29 in leap years), March (31), April (30), May (31), June (30), July (31), August (31), September (30), October (31), November (30), December (31).

NOTE — In certain applications a month is regarded as a period of 30 days.

3.13 period: A duration of time, specified

a) as a defined length of time (e.g. hours, days, months, years);

b) by its beginning and end points.

3.14 second: A basic unit of measurement of time in the International System of Units (SI) as defined in ISO 31-1.

3.15 truncated representation: The abbreviation of a complete representation by omission of higher order components starting from the extreme left-hand side of the expression.

3.16 week: A period of time of seven days.

3.17 week, calendar: A seven day period within a calendar year, starting on a Monday and identified by its ordinal number within the year; the first calendar week of the year is the one that includes the first Thursday of that year. In the Gregorian calendar, this is equivalent to the week which includes 4 January.

3.18 year: A period of time of twelve consecutive months, considered to equal a calendar year.

3.19 year, calendar: A cyclic period of time in a calendar which is required for one revolution of the earth around the sun. In the Gregorian calendar, a calendar year is either a common year or a leap year.

3.20 year, common: In the Gregorian calendar, a year which has 365 days.

3.21 year, leap: In the Gregorian calendar, a year which has 366 days. A leap year is a year whose number is divisible by four an integral number of times, except that if it is a centennial year it shall be divisible by four hundred an integral number of times.

4 Fundamental principles

4.1 Concept

A precise point in calendar time can be identified by means of a unique expression giving a specific day and a specific time within that day. The degree of precision required for the application can be obtained by including the appropriate components.

4.2 Common features, uniqueness and combinations

The decreasing order of components, left-to-right, is common to the expressions for

- precise points in time;
- dates only;
- times only;
- periods of time;
- any abbreviations of the above.

4.3 Characters used in the representations

The representations specified in this International Standard use digits, alphabetic characters and special characters specified in ISO 646. The particular use of these characters is explained in 4.4 and clause 5.

NOTE – Where the upper case characters are not available lower case characters may be used.

The space character shall not be used in the representations.

4.4 Use of separators

When required, the following characters shall be used as separators:

[-] (hyphen) – to separate the time elements "year" and "month", "year" and "week", "year" and "day", "month" and "day", and "week" and "day";

NOTE – The hyphen is also used to indicate omitted components.

[:] (colon) – to separate the time elements "hour" and "minute", and "minute" and "second".

[/] (solidus) – to separate the two components in the representation of periods of time.

4.5 Truncation

It is permitted to omit higher order components (truncation) in applications where their presence is implied. To assure uniqueness of each representation provided for in this International Standard, truncation of a particular representation should be done in accordance with the rules given in the appropriate subclause of clause 5 referring to the representation in question. The addition of a single hyphen in place of each omitted component will usually be necessary, to avoid risk of misinterpretation.

NOTE – By mutual agreement of the partners in information interchange, leading hyphens may be omitted in the applications where there is no risk of confusing these representations with others defined in this International Standard.

4.6 Leading zero(s)

Each date and time component in a defined representation has a defined length, and (a) leading zero(s) shall be used as required.

5 Representations

5.1 Explanations

5.1.1 Characters used in place of digits

[C] represents a digit used in the thousands and hundreds components (the "century" component) of the time element "year";

[Y] represents a digit used in the tens and units components of the time element "year";

[M] represents a digit used in the time element "month";

[D] represents a digit used in the time element "day";

[w] represents a digit used in the time element "week";

[h] represents a digit used in the time element "hour";

[m] represents a digit used in the time element "minute";

[s] represents a digit used in the time element "second";

[n] represents digit(s), constituting a positive integer.

5.1.2 Characters used as designators

[P] is used as period designator, preceding a data element which represents a given duration of a period of time;

[T] is used as time designator to indicate the start of the representation of the time of the day in combined date and time of day expressions;

[W] is used as week designator, preceding a data element which represents the ordinal number of a calendar week within the year;

[Z] is used as time-zone designator, immediately (without space) following a data element expressing the time of the day in Coordinated Universal Time (UTC).

In representations of duration of time (5.5.3.2), the following characters are also used as parts of the representation when required:

[Y] [M] [W] [D] [H] [M] [S]

NOTE – In these representations, [M] may be used to indicate "month" or "minute", or both.

5.2 Dates

For ease of comparison, in all the following examples of representations of dates, the date of 12 April 1985 is used as an illustration, as applicable.

5.2.1 Calendar date

In expressions of calendar dates

— **day of the month** (calendar day) is represented by two digits. The first day of any month is represented by [01] and subsequent days of the same month are numbered in ascending sequence;

— **month** is represented by two digits. January is represented by [01], and subsequent months are numbered in ascending sequence;

— **year** is generally represented by four digits; years are numbered in ascending order according to the Gregorian Calendar.

5.2.1.1 Complete representation

When the application clearly identifies the need for an expression only of a calendar date, then the complete representation shall be a single numeric data element comprising eight digits, where [CCYY] represents a calendar year, [MM] the ordinal number of a calendar month within the calendar year, and [DD] the ordinal number of a day within the calendar month.

Basic format: CCYYMMDD

Example: 19850412

Extended format: CCYY-MM-DD

Example: 1985-04-12

5.2.1.2 Representations with reduced precision

If in a given application it is sufficient to express a calendar date with less precision than a complete representation as specified in 5.2.1.1, either two, four or six digits may be omitted, the omission starting from the extreme right-hand side. The resulting representation will then indicate a month, a year or a century, as set out below. When only [DD] are omitted, a separator shall be inserted between [CCYY] and [MM], but separators shall not be used in the other representations with reduced precision.

a) A specific month

Basic format: CCYY-MM

Example: 1985-04

Extended format: not applicable

b) A specific year

Basic format: CCYY

Example: 1985

Extended format: not applicable

c) A specific century

Basic format: CC

Example: 19

Extended format: not applicable

5.2.1.3 Truncated representations

If truncated representations are required the basic formats shall be as specified below. In each case hyphens (to indicate omitted components) shall be used only as indicated.

a) A specific date in the current century

Basic format: YYMMDD

Example: 850412

Extended format: YY-MM-DD

Example: 85-04-12

b) A specific year and month in the current century

Basic format: -YYMM

Example: -8504

Extended format: -YY-MM

Example: -85-04

c) A specific year in the current century

Basic format: -YY

Example: -85

Extended format: not applicable

d) A specific day of a month

Basic format: --MMDD

Example: --0412

Extended format: --MM-DD

Example: --04-12

e) A specific month

Basic format: --MM

Example: --04

Extended format: not applicable

f) A specific day

Basic format: ---DD

Example: ---12

Extended format: not applicable

5.2.2 Ordinal date

The ordinal day of the year is represented by three decimal digits. The first day of any year is represented by [001] and subsequent days are numbered in ascending sequence.

5.2.2.1 Complete representation

When the application clearly identifies the need for a complete representation of an ordinal date, it shall be one of the numeric expressions as follows, where [CCYY] represents a calendar year and [DDD] the ordinal number of a day within the year.

Basic format: CCYYDDD

Example: 1985102

Extended format: CCYY-DDD

Example: 1985-102

5.2.2.2 Truncated representations

If truncated representations are required, the basic formats shall be as specified below. In each case hyphens (to indicate omitted components) shall be used only as indicated.

a) A specific year and day in the current century

Basic format: YYDDD

Example: 85102

Extended format: YY-DDD

Example: 85-102

b) Day only

Basic format: -DDD

Example: -102

Extended format: not applicable

NOTE — Logically, the representation should be [--DDD], but the first hyphen is superfluous and, therefore, it has been omitted.

5.2.3 Date identified by calendar week and day numbers

Calendar week is represented by two numeric digits. The first calendar week of a year shall be identified as [01] and subsequent weeks shall be numbered in ascending sequence.

Day of the week is represented by one decimal digit. Monday shall be identified as day [1] of any calendar week, and subsequent days of the same week shall be numbered in ascending sequence to Sunday (day [7]).

5.2.3.1 Complete representation

When the application clearly identifies the need for a complete representation of a date identified by calendar week and day numbers, it shall be one of the alphanumeric expressions as follows, where [CCYY] represents a calendar year, [W] is the week designator, [ww] represents the ordinal number of a calendar week within the year, and [D] represents the ordinal number of a day within the calendar week.

Basic format: CCYYWwwD

Example: 1985W155

Extended format: CCYY-Www-D

Example: 1985-W15-5

5.2.3.2 Representation with reduced precision

If the degree of precision required permits, one digit may be omitted from the representation in 5.2.3.1.

Basic format: CCYYWww

Example: 1985W15

Extended format: CCYY-Www

Example: 1985-W15

5.2.3.3 Truncated representations

If truncated representations are required the basic formats shall be as specified below. In each case hyphens (to indicate omitted components) shall be used only as indicated.

a) Year, week and day in the current century

Basic format: YYWwwD

Example: 85W155

Extended format: YY-Www-D

Example: 85-W15-5

b) Year and week only in the current century

Basic format: YYWww

Example: 85W15

Extended format: YY-Www

Example: 85-W15

c) Year of the current decade, week and day only

Basic format: -YWwwD

Example: -5W155

Extended format: -Y-Www-D

Example: -5-W15-5

d) Week and day only of the current year

Basic format: -WwwD

Example: -W155

Extended format: -Www-D

Example: -W15-5

e) Week only of the current year

Basic format: -Www

Example: -W15

Extended format: not applicable

f) Day only of the current week

Basic format: -W-D

Example: -W-5

Extended format: not applicable

NOTE — Although the representation [-W-D] could be abbreviated to [-D] without risk of misinterpretation, the full, logical, derivation has been retained because the [W] serves to identify the representation as a date based on week and day numbers. Its frequency of use is expected to be low and, therefore, the two potentially superfluous characters are not likely to create transmission problems.

g) Day only of any week

Basic format: ---D

Example: ---5

Extended format: not applicable

5.3 Time of the day

As this International Standard is based on the 24-hour timekeeping system which is now in common use, hours are represented by two digits from [01] to [24], whereas minutes and seconds are represented by two digits from [01] to [60]. For most purposes times will be represented by four digits [hhmm].

5.3.1 Local time of the day

5.3.1.1 Complete representation

When the application clearly identifies the need for an expression only of a time of the day then the complete representation shall be a single numeric data element comprising six digits in the basic format, where [hh] represents hours, [mm] minutes and [ss] seconds.

Basic format: hhmmss

Example: 232050

Extended format: hh:mm:ss

Example: 23:20:50

5.3.1.2 Representations with reduced precision

If the degree of precision required permits, either two or four digits may be omitted from the representation in 5.3.1.1.

Basic format: hhmm
hh

Example: 2320
23

Extended format: hh:mm
not applicable

Example: 23:20

5.3.1.3 Representation of decimal fractions

If necessary for a particular application a decimal fraction of hour, minute or second may be included. If a decimal fraction is included, lower order components (if any) shall be omitted, and the decimal fraction shall be divided from the integer part by the decimal sign specified in ISO 31-0: i.e. the comma [,] or full stop [.]. Of these, the comma is the preferred sign. If the magnitude of the number is less than unity, the decimal sign shall be preceded by a zero (see ISO 31-0).

The number of digits in the decimal fraction shall be determined by the interchange parties, dependent upon the application. The format shall be [hhmmss,s], [hhmm,m] or [hh,h] as appropriate (hour minute second, hour minute and hour, respectively), with as many digits as necessary following the decimal sign. If the extended format is required, separators may be included in the decimal representation when the complete representation is used, or when it is reduced by omission of [ss,s].

Basic format: hhmmss,s
hhmm,m
hh,h

Example: 232050,5
2320,9
23,3

Extended format: hh:mm:ss,s
hh:mm,m
not applicable

Example: 23:20:50,5
23:20,9

5.3.1.4 Truncated representations

If truncated representations are required the basic formats shall be as specified below. In each case hyphens (to indicate omitted components) shall be used only as indicated.

a) A specific minute and second of the hour

Basic format: -mmss

Example: -2050

Extended format: -mm:ss

Example: -20:50

b) A specific minute of the hour

Basic format: -mm

Example: -20

Extended format: not applicable

c) A specific second of the minute

Basic format: --ss

Example: --50

Extended format: not applicable

d) A specific hour of the day and decimal fraction of the hour

Basic format: hh,h

Example: 11,3

Extended format: not applicable

e) A specific minute of the hour and a decimal fraction of the minute

Basic format: -mm,m

Example: -20,9

Extended format: not applicable

f) A specific minute and second of the hour and a decimal fraction of the second

Basic format: -mmss,s

Example: -2050,5

Extended format: -mm:ss,s

Example: -20:50,5

g) A specific second of the minute and a decimal fraction of the second

Basic format: --ss,s

Example: --50,5

Extended format: not applicable

NOTE — The basic formats above show only one digit following the decimal sign, but as many digits as required may be used.

5.3.2 Midnight

The complete and extended representations for midnight, in accordance with 5.3.1, shall be expressed in either of the two following ways:

Basic format	*Extended format*
a) 000000	00:00:00 (the beginning of a day);
b) 240000	24:00:00 (the end of a day).

The representations may be reduced in accordance with 5.3.1.4.

NOTES

1 Midnight will normally be represented as [0000] or [2400]

2 The choice of representation a) or b) will depend upon any association with a date, or a time period.

3 The end of one day [2400] coincides with [0000] at the start of the next day, e.g. 2400 on 12 April 1985 is the same as 0000 on 13 April 1985. If there is no association with a date or a time period both a) and b) represent the same clock time in the 24-hour timekeeping system.

5.3.3 Coordinated Universal Time (UTC)

To express the time of the day in Coordinated Universal Time, the representations specified in 5.3.1 shall be used, followed immediately, without spaces, by the time-zone designator [Z]. The examples below are complete and reduced precision representations of the UTC time 20 minutes and 30 seconds past 23 hours:

Basic format: hhmmssZ
hhmmZ
hhZ

Example: 232030Z
2320Z
23Z

Extended format: hh:mm:ssZ
hh:mmZ
not applicable

Example: 23:20:30Z
23:20Z

5.3.3.1 Differences between local time and Coordinated Universal Time

When it is required to indicate the difference between local time and Coordinated Universal Time, its representation shall be appended to the representation of the local time following immediately, without space, the lowest order (extreme right-hand) component of the local time expression, which, in this case, shall always include hours.

The difference between local time and Coordinated Universal Time shall be expressed in hours and minutes, or hours only independently of the precision of the local time expression associated with it. It shall be expressed as positive (i.e. with the leading plus sign [+]) if the local time is ahead of and as negative (i.e. with the leading minus sign [−]) if it is behind Coordinated Universal Time as shown below. The complete representation of the time of 27 minutes 46 seconds past 15 hours locally in Geneva (normally one hour ahead of UTC), and in New York (five hours behind UTC), together with the indication of the difference between the local time and Coordinated Universal Time, are used as examples.

Basic format: +hhmm
+hh
−hhmm
−hh

Example: 152746+0100
152746+01
152746−0500
152746−05

Extended format: +hh:mm
not applicable
−hh:mm
not applicable

Example: 15:27:46+01:00
15:27:46+01
15:27:46−05:00
15:27:46−05

NOTE — The representations of the negative difference between local time and Coordinated Universal Time should not be used alone as they may be confused with the truncated representations of dates provided for in 5.2.1.3, and with truncated representations of time of the day provided for in 5.3.1.4.

5.4 Combinations of date and time of the day representations

When the application does not clearly identify the need for only a date expression (see 5.2) or only a time of the day expression (see 5.3), then a moment of time can be identified through a combination of date and time of the day representations provided for in this International Standard.

5.4.1 Complete representation

The components of an instant of time shall be written in the following sequence:

a) For calendar dates:

year - month - day - time designator - hour - minute - second

b) For ordinal dates:

year - day - time designator - hour - minute - second

c) For dates identified by week and day numbers:

year - week designator - week - day - time designator - hour - minute - second

The character [T] shall be used as time designator to indicate the start of the representation of date time of day in combined date and time of day expressions. The hyphen [-] and the colon [:] shall be used, in accordance with 4.4, as separators within the date and time of the day expressions respectively, when required. When any of the date or time components are omitted, the time designator shall always precede the remaining time of day components.

NOTE — By mutual agreement of the partners in information interchange, the character [T] may be omitted in applications where there is no risk of confusing a combined date and time of the day representation with others defined in this International Standard.

The following are examples of complete and reduced representation (in basic and extended format) of combinations of date and time of the day representations:

a) Calendar date and local time of the day

Basic format: CCYYMMDDThhmmss
CCYYMMDDThhmm
CCYYMMDDThh

Examples: 19850412T101530
19850412T1015
19850412T10

Extended format: CCYY-MM-DDThh:mm:ss
CCYY-MM-DDThh:mm
CCYY-MM-DDThh

Examples: 1985-04-12T10:30
1985-04-12T10:15
1985-04-12T10

b) Ordinal date and local time of the day

Basic format: CCYYDDDThhmmss
CCYYDDDThhmm
CCYYDDDThh

Examples: 1985102T235030
1985102T2350
1985102T23

Extended format: CCYY-DDDThh:mm:ss
CCYY-DDDThh:mm
CCYY-DDDThh

Examples: 1985-102T23:50:30
1985-102T23:50
1985-102T23

c) Date identified by calendar week and day numbers and local time of the day

Basic format: CCYYWwwDThhmmss
CCYYWwwDThhmm
CCYYWwwDThh

Examples: 1985W155T235030
1985W155T2350
1985W155T23

Extended format: CCYY-Www-DThh:mm:ss
CCYY-Www-DThh:mm
CCYY-Www-DThh

Examples: 1985-W15-5T23:30
1985-W15-5T23:50
1985-W15-5T23

5.4.2 Representations other than complete

For reduced precision or truncated representations of a combined date and time expression any of the representations in 5.2.1 (for calendar dates), 5.2.2 (for ordinal dates) or 5.2.3 (for dates identified by week numbers) may be combined with any of the representations in 5.3 provided that the rules specified in those sections are applied, together with the following:

a) the date component shall not be represented with reduced precision and the time component shall not be truncated in a combined date and time expression;

b) when truncation occurs in the date component of a combined date and time expression, it is not necessary to replace the omitted higher order components with the hyphen [-];

c) when the context does not clearly identify a time only component, and if the extended format including colon [:] separator is not used, then it is necessary to commence the time expression with the designator [T].

5.5 *Periods of time*

5.5.1 Means of specifying periods

A period of time shall be expressed in one of the following ways:

a) As a duration of time delimited by a specific start and a specific end;

b) As a quantity of time expressed in one or more specific components but not associated with any specific start or end;

c) As a quantity of time associated with a specific start;

d) As a quantity of time associated with a specific end.

5.5.2 Separators and designators

A solidus [/] shall be used to separate the two components in each of 5.5.1 a), c) and d).

For 5.5.1 b), c) and d) the designator [P] shall precede, without spaces, the representation of the duration.

Other designators (and the hyphen when used to indicate omitted components) shall be used as shown in 5.5.3 below.

NOTE — In certain application areas a double hyphen is used as a separator instead of a solidus.

5.5.3 Complete representations

5.5.3.1 Representation of period identified by its start and end

When the application clearly identifies the need for a complete representation of a period of time, identified by its start and its end, it shall be one of the alphanumeric expressions as set out below. For the specific start or end of a period, [CCYY] represents a calendar year, [MM] the ordinal number of a calendar month within the calendar year, [DD] the ordinal number of a day within the calendar month, [hh] hours, [mm] minutes and [ss] seconds.

Basic format:
CCYYMMDDThhmmss/CCYYMMDDThhmmss

Example: 19850412T232050/19850625T103000

A period beginning at 20 minutes and 50 seconds past 23 hours on 12 April 1985 and ending at 30 minutes past 10 hours on 25 June 1985.

5.5.3.2 Representation of duration of time

A given duration of a period of time, whether or not associated with a start or end, shall be represented by a data element of variable length, preceded by the designator [P]. The number of years shall be followed by the designator [Y], the number of months by [M], the number of weeks by [W], and the number of days by [D]. The part including time components shall be preceded by the designator [T]; the number of hours shall be followed by [H], the number of minutes by [M] and the number of seconds by [S]. In the example set out below, [n] represents one or more digits, constituting a positive integer.

Basic format: PnYnMnDTnHnMnS

PnW

Example: P2Y10M15DT10H30M20S

A duration of two years, 10 months, 15 days, 10 hours, 30 minutes and 20 seconds.

P6W

A period of six weeks.

5.5.3.2.1 Alternative format

If required for particular reasons, durations of time may be expressed in conformity with the format used for points-in-time, as specified in clause 5. Accordingly, the values expressed must not exceed the "carry-over points" of 12 months, 30 days, 24 hours, 60 minutes and 60 seconds. Since weeks have no defined carry-over point (52 or 53), weeks should not be used in these applications.

5.5.3.3 Representation of period identified by its start and its duration

Basic format:
CCYYMMDDThhmmss/PnYnMnDTnHnMnS

Example: 19850412T232050/P1Y2M15DT12H30M

A period of one year, 2 months, 15 days, 12 and a half hours, beginning on 12 April 1985 at 20 minutes and 50 seconds past 23 hours.

5.5.3.4 Representation of period identified by its duration and its end

Basic format:
PnYnMnDTnHnMnS/CCYYMMDDThhmmss

Example: P1Y2M15DT12H30M/19850412T232050

A period of one year, 2 months, 15 days and 12 and a half hours, ending on 12 April 1985 at 20 minutes and 50 seconds past 23 hours.

NOTES

1 Where complete representations using calendar dates have been shown, ordinal dates (5.2.2) or dates identified by week number (5.2.3) may be substituted in similar fashion.

2 In 5.5.3.2, 5.5.3.3 and 5.5.3.4 the components for duration would frequently be in reduced precision form.

If extended formats are required, they shall conform to the requirements of 5.2.1.1, 5.2.2.1, 5.2.3.1 and 5.3.1.1.

5.5.4 Representations other than complete

If reduced precision, or truncated, or decimal representations, or extended formats, are used in place of any components in the complete representations, they shall each be in accordance with the corresponding rules in 5.2 and 5.3.

In representation for the periods in 5.5.1 a),

– if higher order components are omitted from the expression following the solidus (i.e. the representation for "end of period"), it shall be assumed that the corresponding components from the "start of period" expression apply (e.g. if [CCYYMM] are omitted by using a derived representation, the end of the period is in the same year and month as the start of the period);

– representations for time-zones and Coordinated Universal Time included with the component preceding the solidus shall be assumed to apply to the component following the solidus, unless a corresponding alternative is included.

Annex A

Relationship to ISO 2014, 2015, 2711, 3307 and 4031

(This annex does not form part of the standard.)

A.1 In preparing the first edition of ISO 2014 an examination was carried out of the potential uses of all-numeric dates. The advantages of the descending order year-month-day were found to outweigh those for the ascending order day-month-year already established at that time in many parts of the world.

The advantages of the descending order were found to include the following, in particular:

a) the avoidance of confusion in comparison with existing national conventions using different systems of ascending order;

b) the ease with which the whole date may be treated as a single numeral for the purposes of filing and classification;

c) arithmetic calculation, particularly in some computer uses;

d) the possibility of continuing the order by adding digits for hour-minute-second.

A.2 For times, use of the 24-hour timekeeping system is now so common (particularly in view of the wide availability and use of digital watches) that separators to aid human interpretation are no longer necessary but are included as options.

The natural addition of the lower order time digits to the higher order date digits (see above) established the basic concept used in the preparation of this International Standard: that a point in time could be uniquely represented in all-numeric form by a string of digits commencing with year and ending with hour, minute or second, depending on the precision desired.

From that concept representations of all other date and time values were logically derived and, thus, ISO 2014, ISO 3307 and ISO 4031 have been superseded.

A.3 Numbering of days and weeks in the year based on the Gregorian calendar is important in many commercial applications. Methods of numbering the weeks of the year vary from country to country, and, therefore, for international trade and for industrial planning within international companies it is essential to use uniform numbering of weeks. ISO 2015 and ISO 2711 were prepared to meet these requirements.

The uniform numbering of weeks necessitates a unique designation of the day on which a week begins. For commercial purposes, i.e. accounting, planning and similar purposes for which a week number might be used, Monday has been found the most appropriate as the first day of the week.

Identification of a particular date by means of ordinal dates (ISO 2711) and by means of the week numbering system (ISO 2015) were alternative methods that the basic concept of this International Standard could also encompass and, thus, ISO 2015 and ISO 2711 have now been superseded.

Annex B

Examples of representation of dates, time of the day, combinations of date and time, and periods of time

(This annex does not form part of the standard.)

B.1 Dates

Basic format	Extended format	Explanations
Calendar date — 12 April 1985		
19850412	1985-04-12	Complete
850412	85-04-12	Year of any century, with month and date only
--0412	--04-12	Month and date of any year
---12	not applicable	Day only of any month
Ordinal date — 12 April 1985		
1985102	1985-102	Complete
85102	85-102	Year of any century, with ordinal day
5-102	not applicable	Year of any decennium, with ordinal day
-102	not applicable	Ordinal day of any year
Calendar week and day — Friday 12 April 1985		
1985W155	1985-W15-5	Complete
85W155	85-W15-5	Year of any century, with week and day
-5W155	-5-W15-5	Year of any decennium, with week and day
-W155	-W15-5	Week and day of any year
-W-5	not applicable	Any week and day of that week
Calendar week — 15th week of 1985		
1985W15	1985-W15	Complete
85W15	85-W15	Year of any century and week of that year
-5W15	-5W15	Year of any decennium and week of that year
-W15	not applicable	Specific week of any year
Day of the week — Friday		
---5	not applicable	Any Friday
Calendar month — April 1985		
1985-04	not applicable	Complete
-8504	-85-04	Year of any century and month of that year
--04	not applicable	Specific month of any year
Calendar year — 1985		
1985	not applicable	Complete
-85	not applicable	Specific year of any century

B.2 Time of the day

Basic format	Extended format	Explanations
Local time of the day		
27 minutes 46 seconds past 15 hours locally		
152746	15:27:46	Complete
-2746	-27:46	Specific minute and second of any hour
--46	not applicable	Specific second of any minute
Reduced to hours and minutes		
1527	15:27	Complete
-27	not applicable	Specific minute of any hour
Reduced to hours		
15	not applicable	Specific hour of any day
Local time with decimal fractions		
27 minutes 35 and a half seconds past 15 hours locally		
152735,5	15:27:35,5	Complete
-2735,5	-27:35,5	Minute of hour, second with decimal fraction
--35,5	not applicable	Second with decimal fraction of the minute
15,46	not applicable	Hour with decimal fraction of that hour
-27,59	not applicable	Minute with decimal fraction of that minute
-,59	not applicable	Decimal fraction of the minute
--,5	not applicable	Decimal fraction of the second
Midnight — The beginning of a day		
000000	00:00:00	Complete
0000	00:00	Hour and minute only
Midnight — The end of the day		
240000	24:00:00	Complete
2400	24:00	Hour and minute only
Coordinated Universal Time (UTC)		
20 minutes and 30 seconds past 23 hours UTC		
232030Z	23:20:30Z	Complete
2320Z	23:20Z	Hour and minute in UTC
23Z	not applicable	Hour in UTC
Differences between local time and Coordinated Universal Time		
The time of 27 minutes 46 seconds past 15 hours locally in Geneva (one hour ahead of UTC)		
152746+0100	15:27:46+01:00	Complete
152746+01	15:27:46+01	Time difference expressed in hours only
The same time locally in New York (five hours behind UTC)		
152746-0500	15:27:46-05:00	Complete
152746-05	15:27:46-05	Time difference expressed in hours only

B.3 Combinations of date and time

Combinations of calendar date and local time of the day

Basic format	Extended format	Explanations
19850412T101530	1985-04-12T10:15:30	Complete
850412T101530	85-04-12T10:15:30	Within specific year of any century
850412T1015	85-04-12T10:15	Ditto, with hour and minute only
0412T1015	04-12T10:15	Within specific month of any year, with hour and minute only
0412T10	04-12T10	Ditto, with hour only
12T10	12T10	Within specific day of any month, with hour only
850412T10	85-04-12T10	Within specific date of any century, with hour only
12T101530	12T10:15:30	Within specific day of any month, year and century

etc.

Combinations of ordinal date and local time of the day

Basic format	Extended format	Explanations
1985102T235030	1985-102T23:50:30	Complete
85102T235030	85-102T23:50:30	Within specific year of any century
85102T2350	85-102T23:50	Ditto, with hour and minute only
102T2350	102T23:50	Ditto, within specific ordinal date in any year
102T23	102T23	Ditto, with hour only
85102T23	85-102T23	Within specific year of any century, with hour only
102T235030	102T23:50:30	Within specific ordinal date in any year of any century

etc.

Combinations of calendar week, day number and local time of the day

Basic format	Extended format	Explanations
1985W155T235030	1985-W15-5T23:50:30	Complete
85W155T235030	85-W15-5T23:50:30	Within specific year of any century
85W155T2350	85-W15-5T23:50	Ditto, with hour and minute only
W155T2350	W15-5T23:50	Ditto, in any year
W155T23	W15-5T23	Ditto, with hour only
85W155T23	85-W15-5T23	Within specific year of any century, with hour only
W155T235030	W15-5T23:50:30	Within specific week and day of that week, in any century and year

etc.

Combinations of day number and local time of the day

Basic format	Extended format	Explanations
5T235030	5T23:50:30	Any Friday, complete
5T2350	5T23:50	With hour and minute only
5T23	not applicable	With hour only

B.4 Periods of time

Basic format **Extended format**

Period with specific start and specific end

A period beginning at 20 minutes and 50 seconds past 23 hours on 12 April 1985 and ending at 30 minutes past 10 hours on 25 June 1985

Basic format	Extended format
19850412T232050/19850625T103000	1985-04-12T23:20:50/1985-06-25T10:30:00

A period beginning on 12 April 1985 and ending on 25 June 1985

Basic format	Extended format
19850412/0625	1985-04-12/06-25

Duration of a period as a quantity of time

Two years, ten months, 15 days, 10 hours, 20 minutes and 30 seconds

Basic format	Extended format
P2Y10M15DT10H20M30S	not applicable

One year and six months

Basic format	Extended format
P1Y6M	not applicable

Seventy-two hours

PT72H	not applicable

Period with specific start and specific duration

A period of one year, 2 months, 15 days and 12 hours, beginning on 12 April 1985 at 20 minutes and 50 seconds past 23 hours

19850412T232050/P1Y2M15DT12H	1985-04-12T23:20:50/P1Y2M15DT12H

Period of specific duration and with specific end

A period of one year, 2 months, 15 days and 12 hours, ending on 12 April 1985 at 20 minutes and 50 seconds past 23 hours

P1Y2M15DT12H/19850412T232050	P1Y2M15DT12H/1985-04-12T23:20:50

UDC 529 : 003.62

Descriptors : information interchange, documentation, data representation, calendar dates.

Price based on 14 pages

INTERNATIONAL STANDARD ISO 8601 : 1988
TECHNICAL CORRIGENDUM 1

Published 1991-05-01

Data elements and interchange formats — Information interchange — Representation of dates and times

TECHNICAL CORRIGENDUM 1

Éléments de données et formats d'échange — Échange d'information — Représentation de la date et de l'heure
RECTIFICATIF TECHNIQUE 1

Technical corrigendum 1 to International Standard ISO 8601 : 1988 was prepared by Technical Committee ISO/TC 154, *Documents and data elements in administration, commerce and industry.*

Page 6

Subclause 5.3

Lines 3 and 4, delete "[01] to [24]" and "[01] to [60]" and insert "[00] to [24]" and "[00] to [59]"

Subclause 5.3.1.3

Last line, delete "shall be preceded by a zero" and insert "shall be preceded by two zeros in accordance with 4.6"

Page 11

Annex B

Clause B.1, Calendar week — 15th week of 1985, extended format column, delete "-5W15" and insert "-5-W15"

UDC 529 : 003.62 **Ref. No. ISO 8601 : 1988/Cor.1 : 1991 (E)**

Descriptors : information interchange, documentation, data representation, calendar dates

Archival Moving Image Materials

Appendix C: ISO Date Standard

INTERNATIONAL STANDARD ISO 2014

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • ORGANISATION INTERNATIONALE DE NORMALISATION

Writing of calendar dates in all-numeric form

Representation numerique des dates

First Edition – 1976-04-01

UDC 529.2 : 003.35 | Ref. No. ISO 2014-1976 (E)

Descriptors : calendar dates, writing, numeric representation

Forward:

ISO (the International Organization for Standardization) is a worldwide federation of national standards institutes (ISO Member Bodies). The work of developing International Standards is carried out through ISO Technical Committees. Every Member Body interested in a subject for which a Technical Committee has been set up has the right to be represented on the Committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work.

Draft International Standards adopted by the Technical Committee are circulated to the Member Bodies for approval before their acceptance as International Standards by the ISO Council.

Prior to 1972, the results of the work of the Technical Committees were published as ISO Recommendations; these documents are not in the process of being transformed into International Standards. As part of this process, Technical Committee ISO/TC 154 has reviewed ISO Recommendations R 2014 and found it technically suitable for transformation International Standards ISO 2014 therefore replaces ISO Recommendation R 2014 - 1971 to which it is technically identical.

ISO Recommendation R 2014 was approved by the Member Bodies of the following countries:

Austria	Italy	Sri Lanka
Belgium	Japan	Sweden
Canada	Korea,, Dem P. Rep of	Switzerland
Egypt, Arab Rep. of	Korea, Rep. of	Thailand
France	Netherlands	United Kingdom
Germany	Poland	U.S.A.
Greece	Portugal	Yugoslavia
Hungary	South Africa, Rep. of	
India	Spain	

The Member Bodies of the following countries expressed disapproval of the Recommendations on technical grounds:

- Czechoslovakia
- Iraq
- Ireland

No Member Body disapproved the transformation of ISO/R 2014 into an International Standard.

For body of ISO Date Standard, see:

Writing of Calendar dates in all-numeric form

See also:

Archival Moving Image Materials: Contents

Archival Moving Image Materials

Writing of calendar dates in all-numeric form (Appendix C: ISO Date Standard)

0 Introduction:

In all forms of international traffic and exchange, dates must be clearly designated and able to be compared without any ambiguity.

The International Standard for writing of calendar dates in all numeric form has been prepared to obviate the confusion arising from misinterpretation of the significance of the numerals in a date written with numerals only; it is considered that similar confusion does not arise when the month is spelled out, either in full or in abbreviated form.

The occasions on which an all numeric date might be used have been examined and the advantages for these occasions of the descending order year - month - day have been found to outweigh those for the ascending order day - month - year, established in many parts of the world.

The advantages of the descending order include the following in particular:

▶ the ease with which the whole date may be treated as a single number for the purpose of filing and classification (for example for insurance and social security systems).

▶ arithmetic calculation, particularly in some computer uses.

▶ the possibility of continuing the order by adding digits for hour - minute - second.

1 Scope:

The International Standard specifies the writing of dated of the Gregorian calendar in all numeric form, signified by the elements year, month, day.

2 Field of Application:

The International Standard is applicable whenever a calendar date containing the elements year, month, day is written in all numeric form.

3 Rules for Writing calendar Dates:

3.1 Sequences:

An all numeric date shall be written in the following order:

- year - month - day

3.2 Characters:

An all numeric date shall be expressed exclusively in arabic numerals, i.e. by using only the decimal digits 0,1,2,...,9.

3.3 Elements:

An all numeric date shall consist of:

▶ four digits to represent the year

Note: Two digits may be used when no possible confusion can arise from the omission of the century, however, four digits should be applied especially in correspondence and for documentation purposes to indicate clearly that the descending order is used.

▶ two digits to represent the month.

▶ two digits to represent the day.

3.4 Separator:

Where a separator is used in an all-numeric date, only a hyphen or a space shall be used between year and month, and between month and day.

3.5 Examples:

The 1st of July 1976 shall be written in one of the following ways:

a) 19760701

b) 1976-07-01

c) 1976 07 01

See also:

Appendix C: ISO Date Standard

Archival Moving Image Materials

Lisp Machine Manual

Sixth Edition, System Version 99

June 1984

Richard Stallman
Daniel Weinreb
David Moon

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-80-C-0505.

34. Dates and Times

The time package contains a set of functions for manipulating dates and times: finding the current time, reading and printing dates and times, converting between formats, and other miscellany regarding peculiarities of the calendar system. It also includes functions for accessing the Lisp Machine's microsecond timer.

Times are represented in two different formats by the functions in the time package. One way is to represent a time by many numbers, indicating a year, a month, a date, an hour, a minute, and a second (plus, sometimes, a day of the week and timezone). If a year less than 100 is specified, a multiple of 100 is added to it to bring it within 50 years of the present. Year numbers returned by the time functions are greater than 1900. The month is 1 for January, 2 for February, etc. The date is 1 for the first day of a month. The hour is a number from 0 to 23. The minute and second are numbers from 0 to 59. Days of the week are fixnums, where 0 means Monday, 1 means Tuesday, and so on. A timezone is specified as the number of hours west of GMT; thus in Massachusetts the timezone is 5. Any adjustment for daylight savings time is separate from this.

This "decoded" format is convenient for printing out times into a readable notation, but it is inconvenient for programs to make sense of these numbers and pass them around as arguments (since there are so many of them). So there is a second representation, called Universal Time, which measures a time as the number of seconds since January 1, 1900, at midnight GMT. This "encoded" format is easy to deal with inside programs, although it doesn't make much sense to look at (it looks like a huge integer). So both formats are provided; there are functions to convert between the two formats; and many functions exist in two versions, one for each format.

The Lisp Machine hardware includes a timer that counts once every microsecond. It is controlled by a crystal and so is fairly accurate. The absolute value of this timer doesn't mean anything useful, since it is initialized randomly; what you do with the timer is to read it at the beginning and end of an interval, and subtract the two values to get the length of the interval in microseconds. These relative times allow you to time intervals of up to an hour (32 bits) with microsecond accuracy.

The Lisp Machine keeps track of the time of day by maintaining a *timebase*, using the microsecond clock to count off the seconds. On the CADR, when the machine first comes up, the timebase is initialized by querying hosts on the Chaosnet to find out the current time. The Lambda has a calendar clock which never stops, so it normally does not need to do this. You can also set the time base using time:set-local-time, described below.

There is a similar timer that counts in 60ths of a second rather than microseconds; it is useful for measuring intervals of a few seconds or minutes with less accuracy. Periodic housekeeping functions of the system are scheduled based on this timer.

US005806063A

United States Patent [19]

Dickens

[11] Patent Number: 5,806,063

[45] Date of Patent: Sep. 8, 1998

[54] **DATE FORMATTING AND SORTING FOR DATES SPANNING THE TURN OF THE CENTURY**

[75] Inventor: **Bruce Dickens**, Irvine, Calif.

[73] Assignee: **McDonnell Douglas Corporation**, Long Beach, Calif.

[21] Appl. No.: **725,574**

[22] Filed: **Oct. 3, 1996**

[51] **Int. Cl.**^{6} **G06F 17/30**

[52] **U.S. Cl.** **707/6**; 707/102; 707/7; 707/200

[58] **Field of Search** 707/6, 102, 7, 707/200

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,573,127	2/1986	Korff	364/493
5,630,118	5/1997	Shaughnessy	707/1
5,644,762	7/1997	Soeder	707/6
5,668,989	9/1997	Mao	707/101

OTHER PUBLICATIONS

The Year 2000 and 2–Digit Dates: A Guide for Planning and Implementation, Third Edition, May, 1996.

IBM: *The Year 2000 and 2–Digit Dates: A Guide for Planning and Implementation*; First Edition, Oct. 1995.

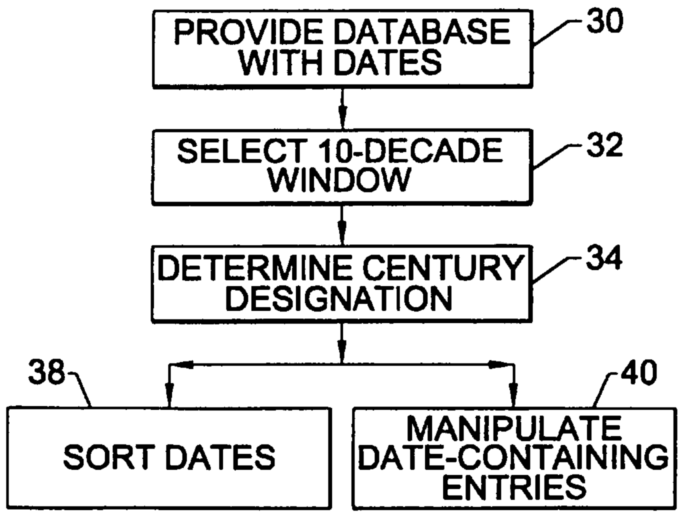
Primary Examiner—Wayne Amsbury

Attorney, Agent, or Firm—Bell Seltzer Intellectual Property Group of Alston & Bird LLP

[57] **ABSTRACT**

Dates stored in symbolic form in a database are reformatted to permit easy manipulation and sorting of date-related information. Each date in M_1M_2, D_1D_2, and Y_1Y_2 format is converted to C_1C_2, Y_1Y_2, M_1M_2, and D_1D_2 format. To accomplish the conversion, a 10-decade window starting on Y_AY_B is defined that encompasses all dates in the database. The value of C_1C_2 is determined by the relative values of Y_1Y_2 and Y_AY_B. The reformatted date information is particularly useful when the reformatting is in $C_1C_2Y_1Y_2M_1M_2D_1D_2$ format, because sorting by date is accomplished using a pure numerical-value sort.

15 Claims, 1 Drawing Sheet


PROVIDE DATABASE WITH DATES
30
SELECT 10-DECADE WINDOW
32
DETERMINE CENTURY DESIGNATION
34
38
SORT DATES
40
MANIPULATE DATE-CONTAINING ENTRIES

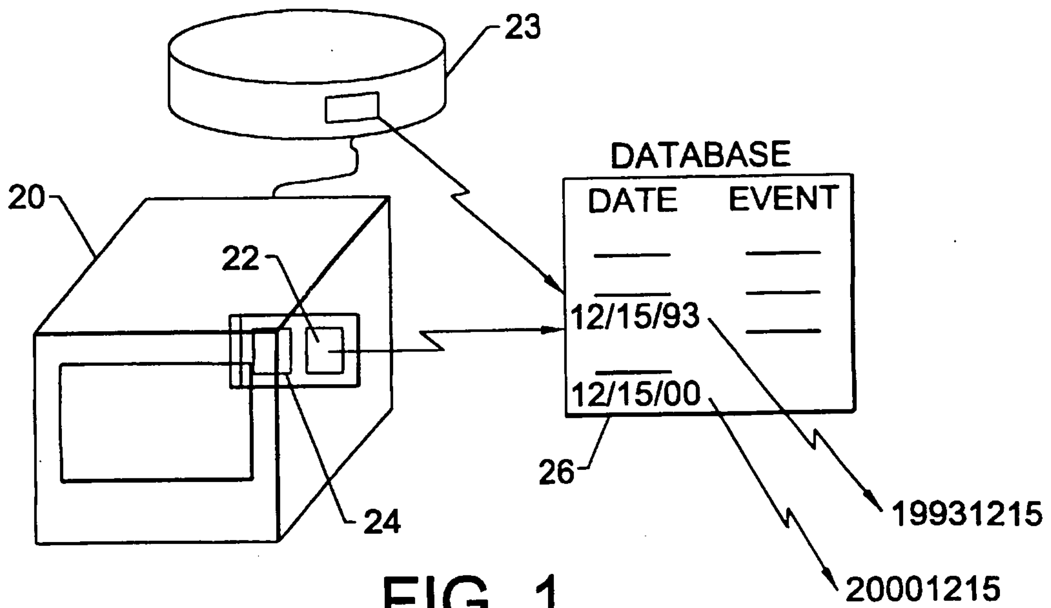

23
20
22
24
DATABASE
DATE EVENT
12/15/93
12/15/00
26
19931215
20001215

FIG. 1.

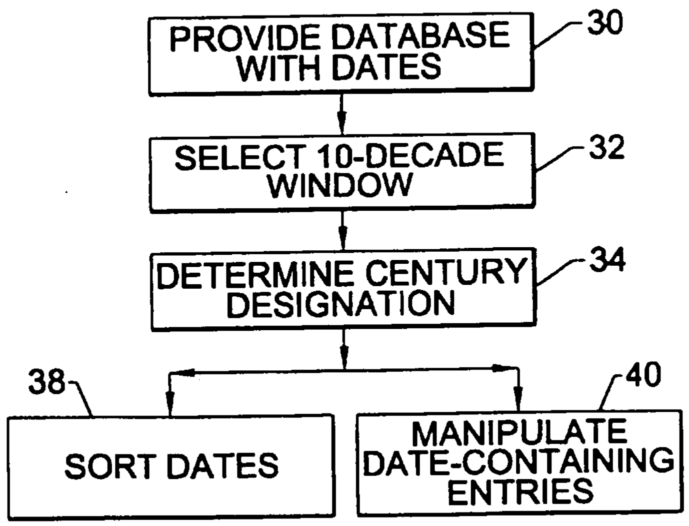

PROVIDE DATABASE WITH DATES
30
SELECT 10-DECADE WINDOW
32
DETERMINE CENTURY DESIGNATION
34
38
SORT DATES
40
MANIPULATE DATE-CONTAINING ENTRIES

FIG. 2.

DATE FORMATTING AND SORTING FOR DATES SPANNING THE TURN OF THE CENTURY

BACKGROUND OF THE INVENTION

This invention relates to the manipulation of information in a database, and, in particular, to the determination of dates in a useful form.

Dates are stored as symbolic representations in computer databases in varying formats. For example, a date may be represented in the numerical representation MM/DD/YY, where MM is a two-digit month designator, DD is a two-digit day designator, and YY is a two-digit year designator (the last two digits of the year). Thus, Dec. 15, 1993 is designated as 12/15/93. A date may also be represented in an alphanumeric form MMM/DD/YY, where MMM is an alphabetic month designator (e.g., DEC for December), and DD and YY are the same as in the numerical form. Dec. 15, 1993 is represented in this format as DEC/15/93.

Such approaches for the representation of dates have worked well since the advent of computer databases, which has occurred in the twentieth century. Dates may be sorted in chronological order using the numerical representations. However, with the turn of the century at Jan. 1, 2000, the representation and utilization of dates becomes more complex. Using the numerical form above, Dec. 15, 2000 is represented as 12/15/00. If a numerical sort is performed on 12/15/93 and 12/15/00, the later date 12/15/00 sorts as the first-occurring date, an incorrect result.

Sets of dates spanning the turn of the century and associated with past, current, and future activities are now stored in many databases. When stored in the conventional formats discussed above, those dates will not readily be used and numerically sorted in chronological order. They may be manually converted to a more usable form in the sense that programs may be written to perform conversions, manipulations, and sorting. However, these programs typically require additional data fields for storage, which may be objectionable in some circumstances.

There is a need for an improved approach to the representation and utilization of dates in databases, and for converting the existing dates in databases to a more usable form. The present invention fulfills this need, and further provides related advantages.

SUMMARY OF THE INVENTION

The present invention provides an approach to the representation and utilization of dates stored symbolically in databases. Existing symbolic date representations are converted to a more useful form of symbolic date representations without the addition of new data fields, and in a manner that is performed automatically by the computer and requires no user input. The approach of the invention permits direct numerical sorting of dates.

In accordance with the invention, a method of processing dates stored in a database comprises the steps of providing a database with dates stored therein according to a format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator, and Y_1Y_2 is the numerical year designator, all of the dates falling within a 10-decade period of time. A 10-decade window with a Y_AY_B value for the first year of the ten-decade window is selected, Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database. A century designator C_1C_2 is determined for each date in the database, C_1C_2 having a first value if Y_1Y_2 is less

than Y_AY_B and having a second value if Y_1Y_2 is equal to or greater than Y_AY_B Each date in the database is formatted with the values C_1C_2, Y_1Y_2, M_1M_2, and D_1D_2.

In the case of most practical interest, the 10-decade period of time spans the year 2000 and begins with a year in which the second digit (Y_B in Y_AY_B) is 0 (zero). For any 10-decade period including the year 2000, if the decade designator Y_1 of the date in the database is numerically less than the decade designator Y_A of the first decade of the 10-decade period of time, the century designator C_1C_2 is "20". If Y_1 is equal to or greater than Y_A, C_1C_2 is "19". Dates in databases spanning more than 10 decades are not handled by this approach, but it is not expected that this limitation will be significant for most commercial and industrial databases.

This approach works particularly well if the dates are represented in the format $C_1C_2Y_1Y_2M_1M_2D_1D_2$. The date Dec. 15, 2000 is represented in this format as 20001215, for example. Dates represented in this format may be directly sorted numerically by fast sorting techniques, and thereafter stored back in the database.

The present invention thus provides an efficient approach to converting and utilizing symbolic date representations in databases, which allows automatic processing of dates ranging from before to after the year 2000. The large number of dates represented in some databases may thereby be readily processed and utilized. Other features and advantages of the present invention will be apparent from the following more detailed description of the preferred embodiment, taken in conjunction with the accompanying drawings, which illustrate, by way of example, the principles of the invention. The scope of the invention is not, however, limited to this preferred embodiment.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a schematic representation of a computer database with date information therein; and

FIG. 2 is a block flow diagram of a preferred approach for practicing the approach of the invention.

DETAILED DESCRIPTION OF THE INVENTION

FIG. 1 schematically depicts a computer 20 having a read-only or random-access memory 22, a mass-storage device 23, and a central processing unit 24 therein. Stored in the memory 22 or on the mass-storage device 23 is a database 26. The database includes information in the form of symbolic representations of dates and associated information such as events occurring on the respective dates. In a conventional approach, the dates are stored in a format such as $M_1M_2/D_1D_2/Y_1Y_2$ format. M indicates month information, D day information, and Y year information, with the subscript 1 or 2 indicating the first or second digit of the designator, respectively. Dec. 15, 1993 is stored as 12/15/93 or 12-15-93, and Dec. 15, 2000 is stored as 12/15/00 or 12-15-00, for example. If a numerical sort is performed on these dates, 12/15/00 will sort chronologically prior to 12/15/93.

FIG. 2 illustrates the approach of the invention. The computer database 26 is provided, numeral 30, having symbolic representations of dates stored therein. In some cases, the dates will be represented as discussed in the preceding paragraph. In other cases, an alphanumeric designator is used. In that approach, each date is stored as $M_aM_bM_c/D_1D_2/Y_1Y_2$ format, where $M_aM_bM_c$ is an alphabetical symbol such as JAN for January, FEB for February, etc. In that case, the month designator $M_aM_bM_c$ is first converted to the numerical form M_1M_2 by converting JAN to "01", FEB to "02", etc.

A 10-decade window is selected, numeral **32**. That is, it is necessary that all dates in the database will be within some period of 10 decades, or 100 years. This limitation poses little problem for most industrial and commercial databases. The window may be arbitrarily selected. For example, the decade could begin with the 1950's and end with the 2040's, or it could begin with the 1980's and end with the 2070's. The 10-decade window will normally include some decades from the prior century and some from the new century.

The first year of the 10-decade window is represented by Y_AY_B. In a commonly utilized application, Y_B is 0 (zero), although the invention is not limited to this case. That is, the 1950's first decade would be represented by Y_A0 of "50", and the 1980's first decade would be represented by Y_A0 of "80". For this case, a century designator C_1C_2 for a date is determined, numeral **34**, by comparing the value of Y_1, the first digit of the year designator for the date, with Y_A, the first digit of the first decade of the 10-decade window. C_1C_2 is assigned a first value if Y_1 is less than Y_A and a second value if Y_1 is equal to or greater than Y_A.

In the case of most interest, the 10-decade window includes decades earlier than the year 2000 and decades later than the year 2000, and Y_B is zero C_1C_2 is assigned "20" if Y_1 is less than Y_A and is assigned "19" if Y_1 is equal to or greater than Y_A. In that case and for example, if Y_A is 5, meaning that the decade beginning in 1950 was selected as the first decade of the 10-decade window, and if Y_1Y_2 is "43", the century designator C_1C_2 is "20", indicating that the year in question in the database is 2043. On the other hand, if Y_1Y_2 is "63", the century designator C_1C_2 is "19", indicating that the year in question in the database is 1963. This selection process is performed in a completely automated fashion by the computer, without human input other than to select the starting date of the 10-decade window.

The symbolic representations of the dates in the database are reformatted with the values C_1C_2, Y_1Y_2, M_1M_2, and D_1D_2, numeral **36** of FIG. 2. In one case that produces particularly advantageous results for many operations, such as chronological date sorting, the date is represented in the form $C_1C_2Y_1Y_2M_1M_2D_1D_2$. For example, the date 12/15/93 (Dec. 15, 1993) is represented as 19931215 and the date 12/15/00 (Dec. 15, 2000) as 20001215. A straightforward numerical sort of date data fields expressed in this form produces an accurate chronological ordering.

Once the symbolic representations of the dates are reformatted according to the procedures set forth above, the date information may be sorted, numeral **38**, or otherwise manipulated, numeral **40**, together with the entries associated with the dates. Such manipulation may include handling of data associated with the dates, storing the dates and associated information back in the data base, or other processes.

The approach of the invention has been implemented in a computer program, a copy of which is attached as Exhibit A. This program converts dates both before and after the year 2000.

The present invention provides an effective technique for reformatting symbolic representations of date information that is rapid and automated, and yields new symbolic representations of date information that are particularly amenable to further processing. Although a particular embodiment of the invention has been described in detail for purposes of illustration, various modifications and enhancements may be made without departing from the spirit and scope of the invention. Accordingly, the invention is not to be limited except as by the appended claims.

What is claimed is:

1. A method of processing symbolic representations of dates stored in a database, comprising the steps of
   providing a database with symbolic representations of dates stored therein according to a format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator, and Y_1Y_2 is the numerical year designator, all of the symbolic representations of dates falling within a 10-decade period of time;
   selecting a 10-decade window with a Y_AY_B value for the first decade of the window, Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database;
   determining a century designator C_1C_2 for each symbolic representation of a date in the database, C_1C_2 having a first value if Y_1Y_2 is less than Y_AY_B and having a second value if Y_1Y_2 is equal to or greater than Y_AY_B; and
   reformatting the symbolic representation of the date with the values C_1C_2, Y_1Y_2, M_1M_2, and D_1D_2 to facilitate further processing of the dates.

2. The method of claim 1, wherein the 10-decade window includes the decade beginning in the year 2000.

3. The method of claim 2, wherein the step of determining includes the step of
   determining the first value as 20 and the second value as 19.

4. The method of claim 1, including an additional step, after the step of reformatting, of
   sorting the symbolic representations of dates.

5. The method of claim 1, wherein the step of reformatting includes the step of
   reformatting each symbolic representation of a date into the format $C_1C_2Y_1Y_2M_1M_2D_1D_2$.

6. The method of claim 5, including an additional step, after the step of reformatting, of
   sorting the symbolic representations of dates using a numerical-order sort.

7. The method of claim 1, wherein the step of providing a database includes the step of
   converting pre-existing date information having a different format into the format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator and Y_1Y_2 is the numerical year designator.

8. The method of claim 1, wherein the step of selecting includes the step of
   selecting Y_AY_B such that Y_B is 0 (zero).

9. The method of claim 1, including an additional step, after the step of reformatting, of
   storing the symbolic representation of dates and their associated information back into the database.

10. The method of claim 9, including the additional step, after the step of reformatting, of
   manipulating information in the database having the reformatted date information therein.

11. A method of processing dates in a database, comprising the steps of
   providing a database with dates stored therein according to a format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator, and Y_1Y_2 is the numerical year designator, all of dates falling within a 10-decade period of time which includes the decade beginning in the year 2000;

selecting a 10-decade window with a Y_AY_B value for the first decade of the window, Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database;

determining a century designator C_1C_2 for each date in the database, C_1C_2 having a first value if Y_1Y_2 is less than Y_AY_B and having a second value if Y_1Y_2 is equal to or greater than Y_AY_B;

reformatting each date in the form $C_1C_2Y_1Y_2M_1M_2D_1D_2$ to facilitate further processing of the dates; and

sorting the dates in the form $C_1C_2Y_1Y_2M_1M_2D_1D_2$.

12. The method of claim **11**, wherein the step of providing a database includes the step of

converting pre-existing date information having a different format into the format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator and Y_1Y_2 is the numerical year designator.

13. The method of claim **11**, wherein the step of selecting includes the step of

selecting Y_AY_B such that Y_B is 0 (zero).

14. The method of claim **11**, including an additional step, after the step of sorting, of

storing the sorted dates and their associated information back into the database.

15. The method of claim **14**, including the additional step, after the step of sorting, of

manipulating information in the database having the reformatted date therein.

* * * * *

IDS

United States Patent [19]

Korff

[11] Patent Number: 4,573,127

[45] Date of Patent: Feb. 25, 1986

[54] PROGRAMMABLE ELECTRONIC REAL-TIME LOAD CONTROLLER, AND APPARATUS THEREFOR, PROVIDING FOR UPDATING OF PRESET CALENDAR EVENTS

[75] Inventor: William W. Korff, Seattle, Wash.

[73] Assignee: Butler Manufacturing Company, Kansas City, Mo.

[21] Appl. No.: 452,627

[22] Filed: Dec. 23, 1982

[51] Int. Cl.⁴ G01R 21/00; G06F 15/20; G06F 15/56

[52] U.S. Cl. 364/493; 307/40; 364/138; 368/28

[58] Field of Search 364/483, 492, 464, 705, 364/138; 368/28, 29, 34, 41; 324/116; 40/120, 107, 109; 307/35, 39, 40

[56] References Cited

U.S. PATENT DOCUMENTS

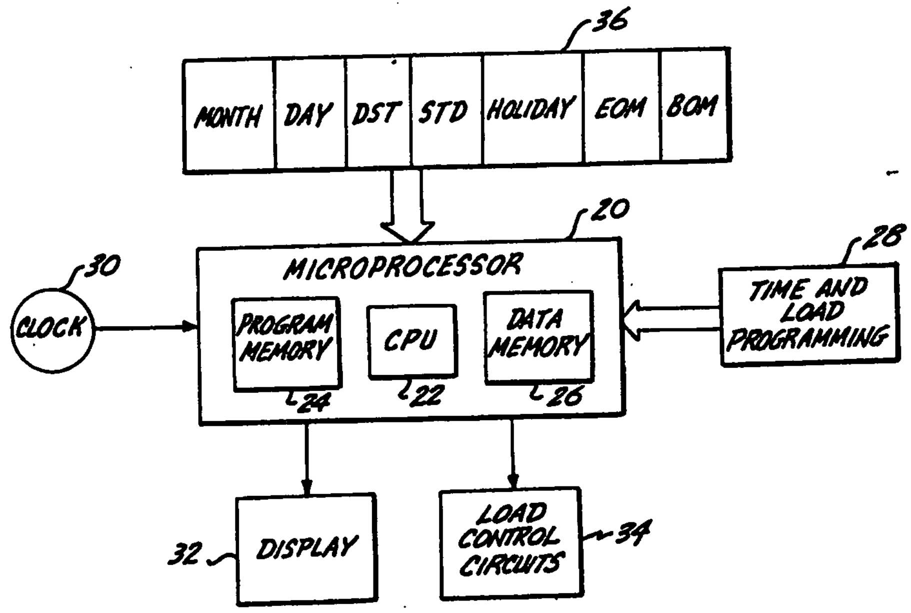
4,274,146	6/1981	Yanagawa	364/705
4,283,772	8/1981	Johnston	364/483
4,291,375	9/1981	Wolf	364/483
4,293,915	10/1981	Carpenter et al.	364/493
4,347,576	8/1982	Kensinger et al.	364/493
4,355,361	10/1982	Riggs et al.	364/483
4,357,665	11/1982	Korff	364/493
4,415,271	11/1983	Mori	368/41

Primary Examiner—Felix D. Gruber
Attorney, Agent, or Firm—Christensen, O'Connor, Johnson & Kindness

[57] ABSTRACT

Certain calendar events, such as the standard/daylight time transition or a holiday such as Labor Day, always occur on the same alphabetic day always having the same relationship to the beginning or end of the same month in any year; however, their numeric day varies from year to year. In order to update the numeric day of such a calendar event that is used by a microprocessor of a programmable electronic real-time load controller, the microprocessor selects a reference day value representing the numeric day of the calendar event in a given year and selects a reference year value representing the numeric year of that year. At the beginning of each real-time year, the microprocessor determines a real-time year value representing the numeric year of the real-time year, determines a current numeric day value for the calendar event from the reference day value, the reference year value, and the real-time year value, and stores the current numeric day value for use during the current real-time year.

17 Claims, 10 Drawing Figures



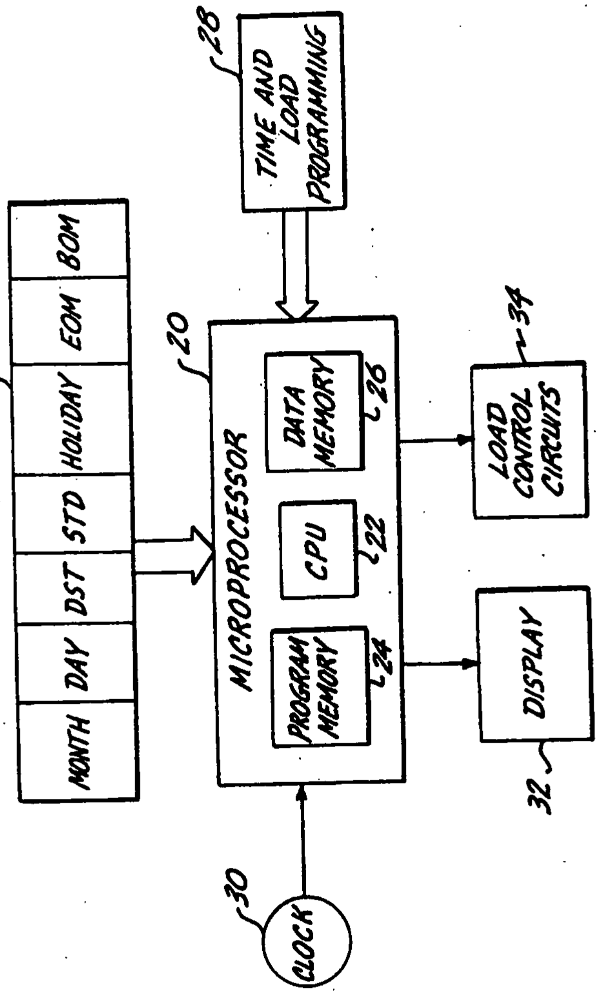

36
MONTH
DAY
DST
STD
HOLIDAY
EOM
BOM
28
TIME AND LOAD PROGRAMMING
20
MICROPROCESSOR
PROGRAM MEMORY
24
CPU
22
DATA MEMORY
26
30
CLOCK
LOAD CONTROL CIRCUITS
34
DISPLAY
32

Fig. 1.

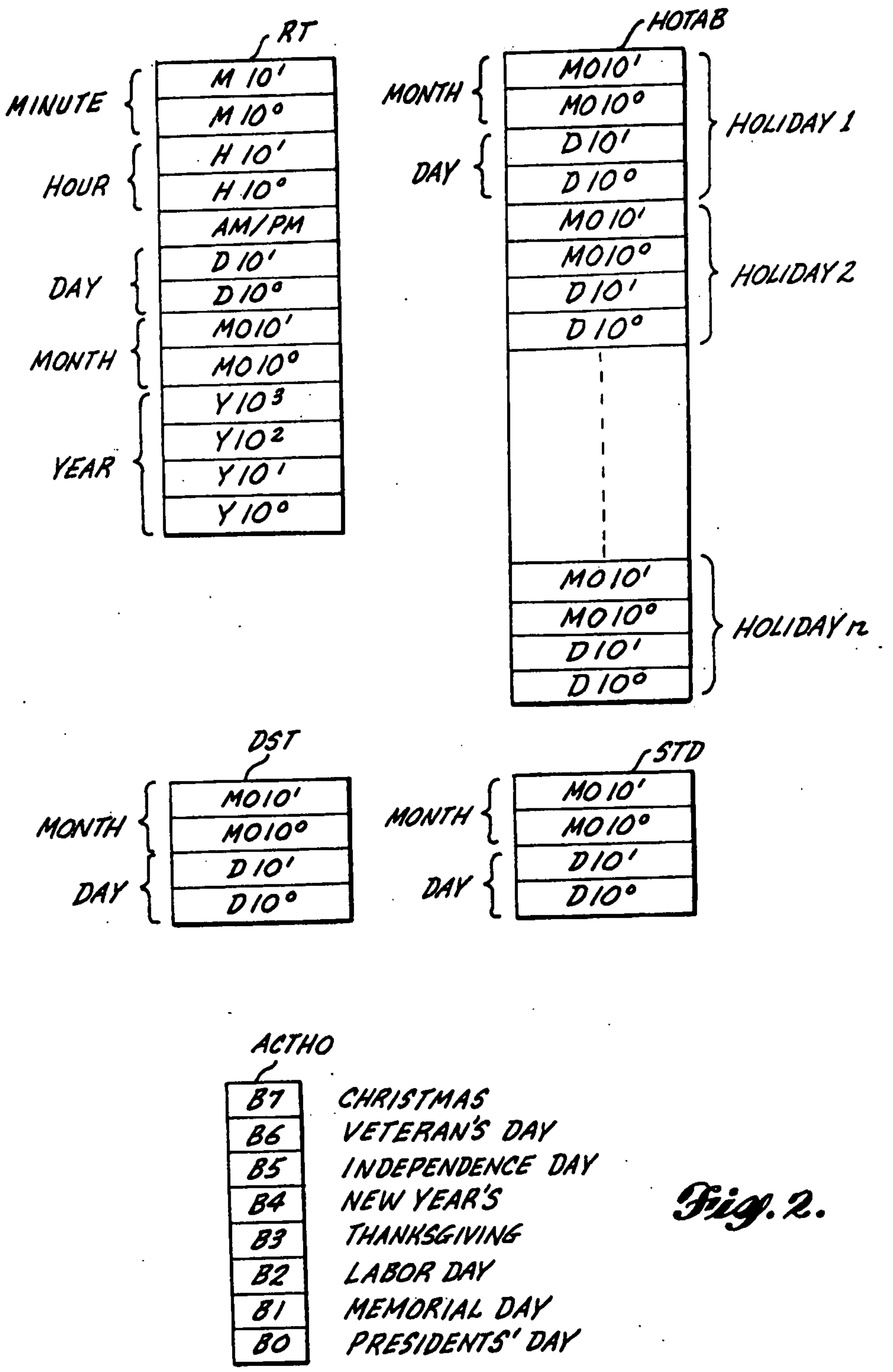

RT
MINUTE
M 10'
M 10°
HOUR
H 10'
H 10°
AM/PM
DAY
D 10'
D 10°
MONTH
MO 10'
MO 10°
YEAR
Y 10³
Y 10²
Y 10'
Y 10°
HOTAB
MONTH
MO 10'
MO 10°
DAY
D 10'
D 10°
HOLIDAY 1
MO 10'
MO 10°
D 10'
D 10°
HOLIDAY 2
MO 10'
MO 10°
D 10'
D 10°
HOLIDAY n
DST
MONTH
MO 10'
MO 10°
DAY
D 10'
D 10°
STD
MONTH
MO 10'
MO 10°
DAY
D 10'
D 10°
ACTHO
B7 CHRISTMAS
B6 VETERAN'S DAY
B5 INDEPENDENCE DAY
B4 NEW YEAR'S
B3 THANKSGIVING
B2 LABOR DAY
B1 MEMORIAL DAY
B0 PRESIDENTS' DAY

Fig. 2.

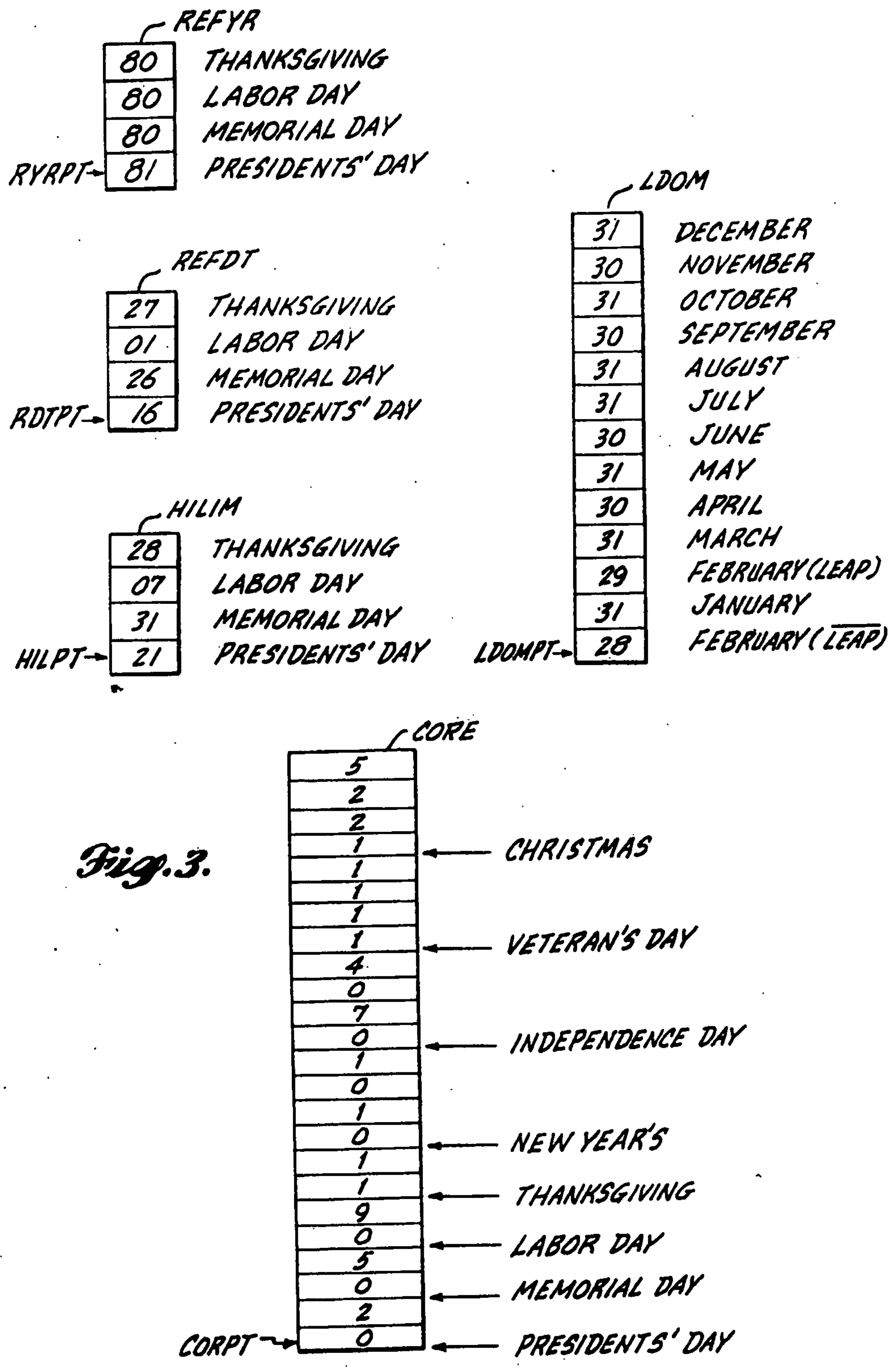

REFYR
80 THANKSGIVING
80 LABOR DAY
80 MEMORIAL DAY
RYRPT 81 PRESIDENTS' DAY
REFDT
27 THANKSGIVING
01 LABOR DAY
26 MEMORIAL DAY
RDTPT 16 PRESIDENTS' DAY
HILIM
28 THANKSGIVING
07 LABOR DAY
31 MEMORIAL DAY
HILPT 21 PRESIDENTS' DAY
LDOM
31 DECEMBER
30 NOVEMBER
31 OCTOBER
30 SEPTEMBER
31 AUGUST
31 JULY
30 JUNE
31 MAY
30 APRIL
31 MARCH
29 FEBRUARY(LEAP)
31 JANUARY
LDOMPT 28 FEBRUARY($\overline{\text{LEAP}}$)
CORE
5
2
2
1 CHRISTMAS
1
1
1
1 VETERAN'S DAY
4
0
7
0 INDEPENDENCE DAY
1
0
1
0 NEW YEAR'S
1
1 THANKSGIVING
9
0 LABOR DAY
5
0 MEMORIAL DAY
2
CORPT 0 PRESIDENTS' DAY

Fig. 3.

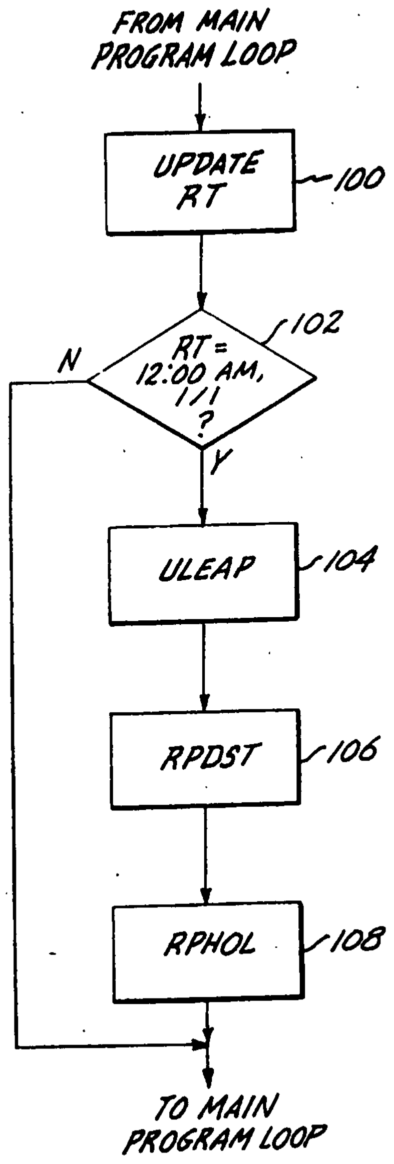

FROM MAIN PROGRAM LOOP
UPDATE RT
100
RT = 12:00 AM, 1/1 ?
102
N
Y
ULEAP
104
RPDST
106
RPHOL
108
TO MAIN PROGRAM LOOP

Fig. 4.

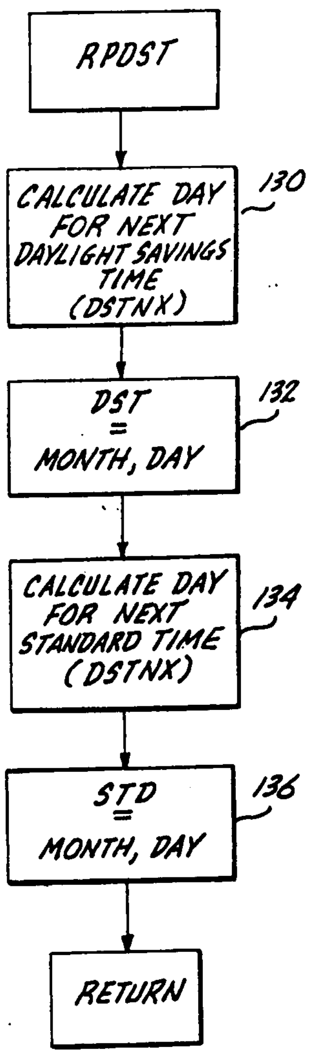
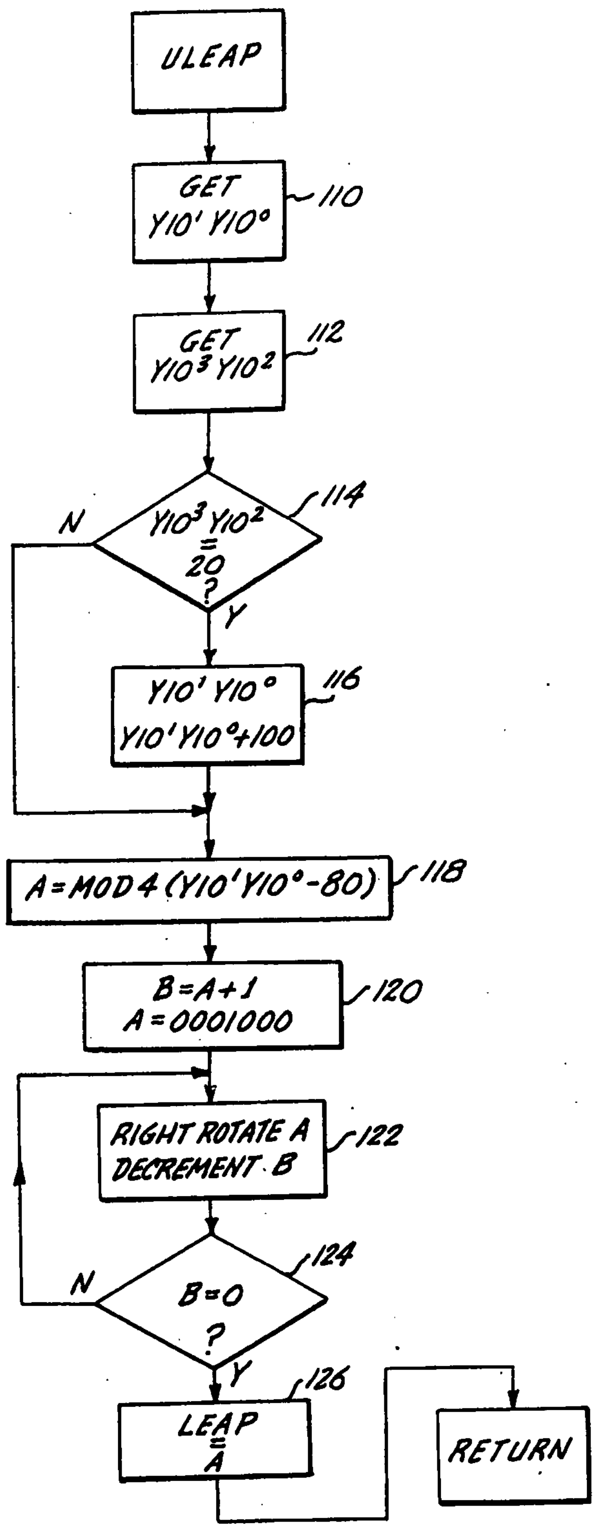

RPDST
CALCULATE DAY FOR NEXT DAYLIGHT SAVINGS TIME (DSTNX)
130
DST = MONTH, DAY
132
CALCULATE DAY FOR NEXT STANDARD TIME (DSTNX)
134
STD = MONTH, DAY
136
RETURN

Fig. 6.

Fig. 5.


ULEAP
GET Y10' Y10°
110
GET Y10³ Y10²
112
Y10³ Y10² = 20 ?
114
N
Y
Y10' Y10° Y10' Y10° +100
116
A = MOD 4 (Y10' Y10° -80)
118
B = A + 1 A = 0001000
120
RIGHT ROTATE A DECREMENT B
122
B = 0 ?
124
N
Y
LEAP = Ā
126
RETURN

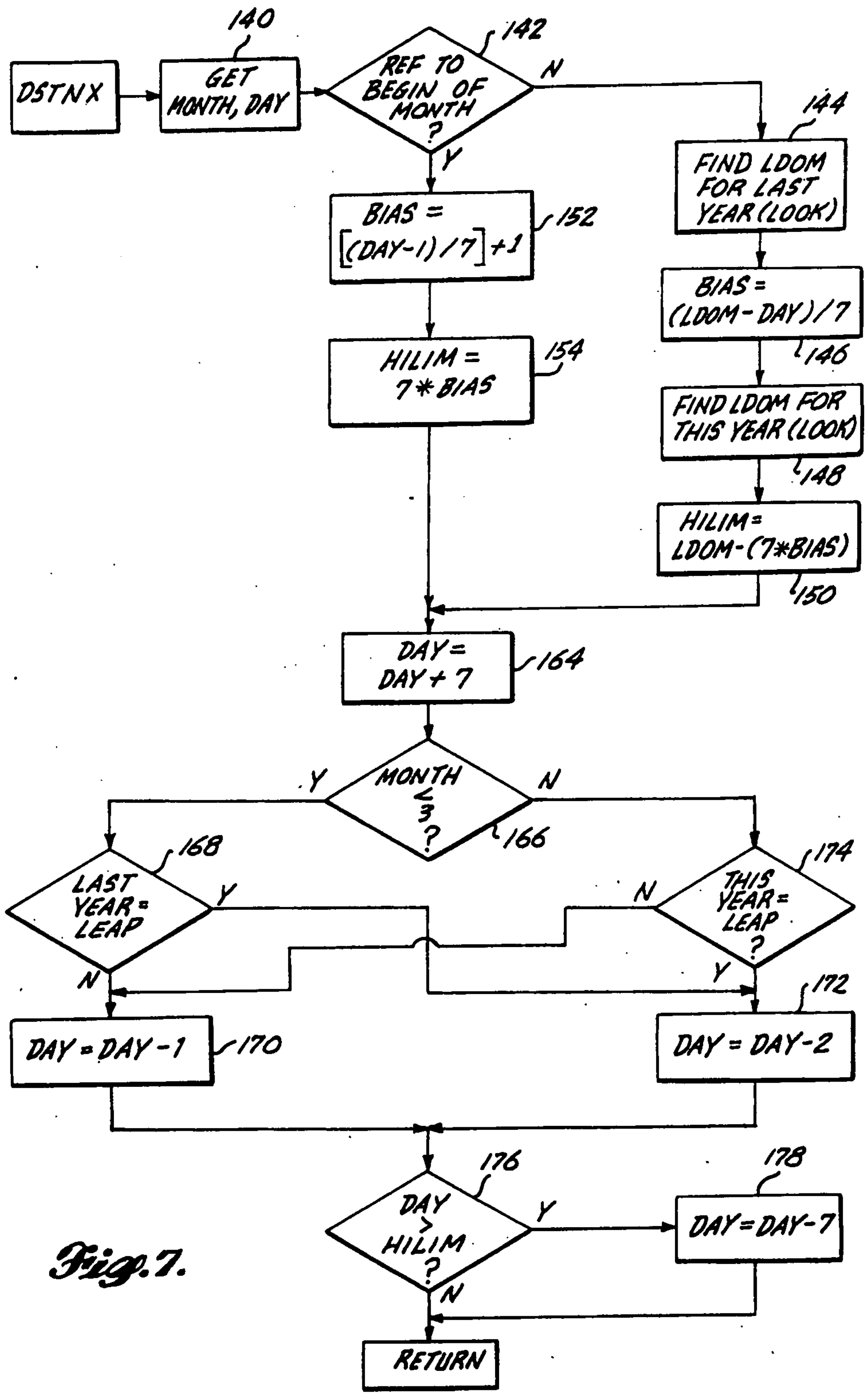

DSTNX
140
GET MONTH, DAY
142
REF TO BEGIN OF MONTH ?
N
Y
144
FIND LDOM FOR LAST YEAR (LOOK)
BIAS = (LDOM - DAY)/7
146
FIND LDOM FOR THIS YEAR (LOOK)
148
HILIM = LDOM-(7*BIAS)
150
BIAS = [(DAY-1)/7]+1
152
HILIM = 7*BIAS
154
DAY = DAY + 7
164
MONTH ≤ 3 ?
166
Y
N
168
LAST YEAR = LEAP
Y
N
174
THIS YEAR = LEAP ?
N
Y
DAY = DAY - 1
170
DAY = DAY - 2
172
176
DAY > HILIM ?
Y
N
178
DAY = DAY - 7
RETURN

Fig. 7.

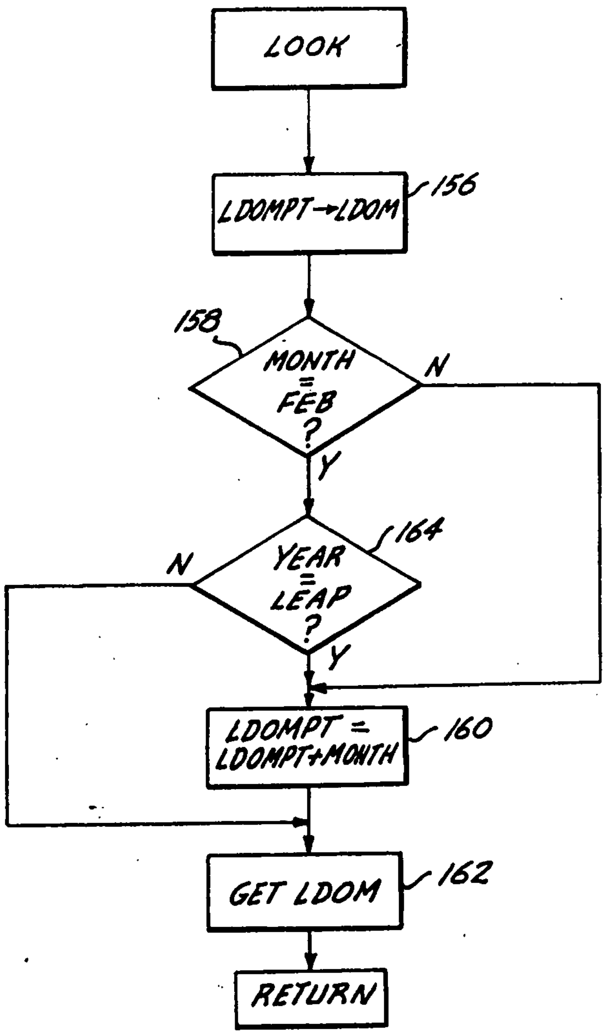

LOOK
LDOMPT→LDOM
156
158
MONTH = FEB ?
N
Y
YEAR = LEAP ?
164
N
Y
LDOMPT = LDOMPT+MONTH
160
GET LDOM
162
RETURN

Fig. 8.

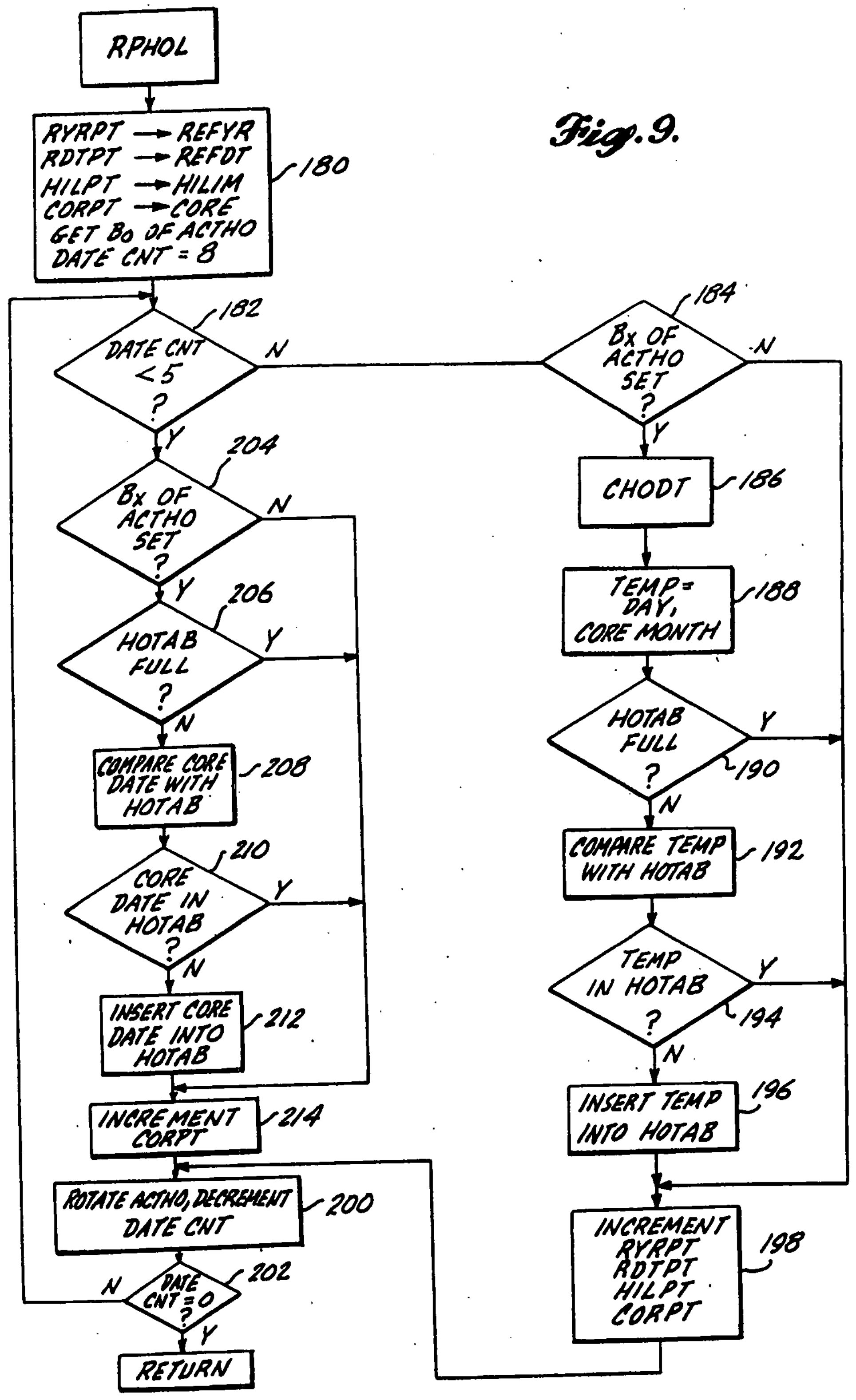
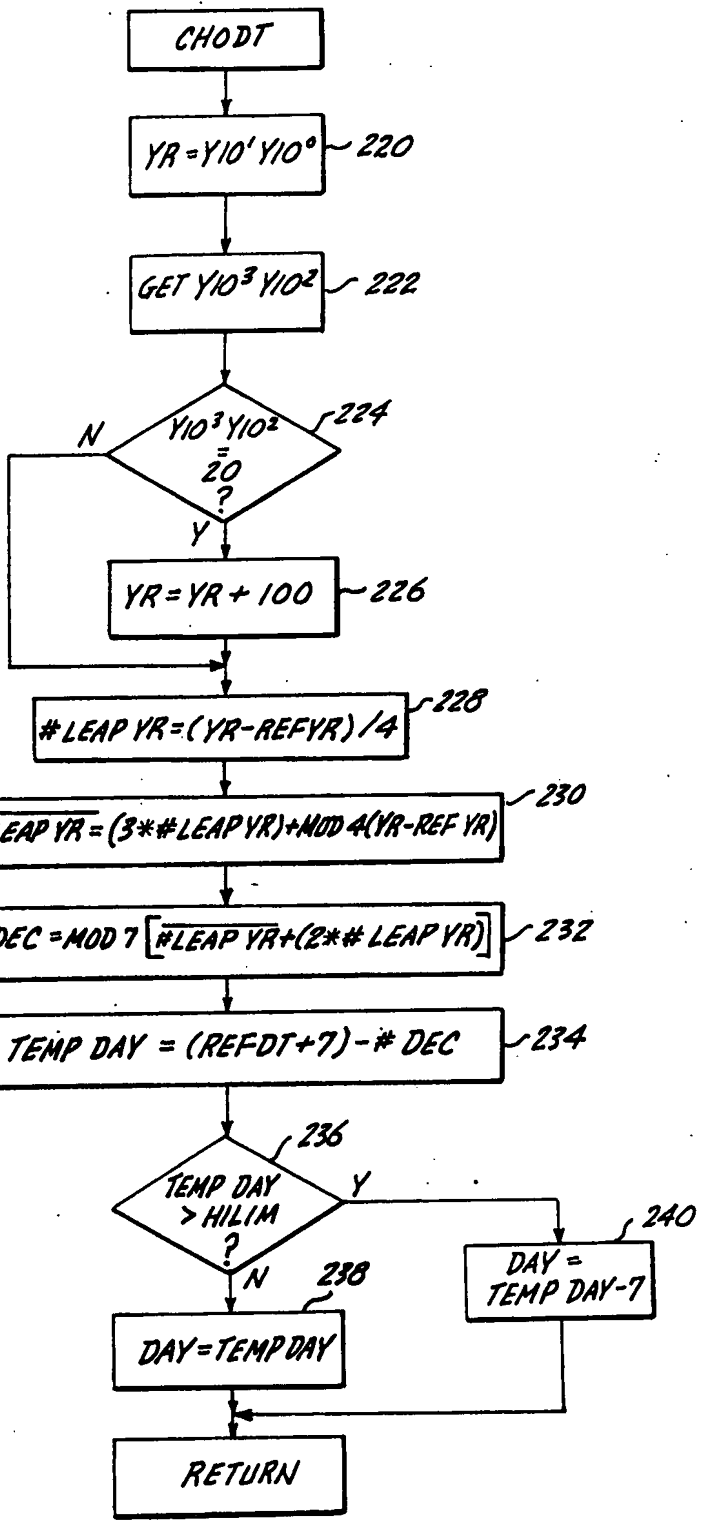

RPHOL
RYRPT → REFYR
RDTPT → REFDT
HILPT → HILIM
CORPT → CORE
GET B0 OF ACTHO
DATE CNT = 8
180
DATE CNT <5 ?
182
N
Y
Bx OF ACTHO SET ?
184
204
206
HOTAB FULL ?
COMPARE CORE DATE WITH HOTAB
208
CORE DATE IN HOTAB ?
210
INSERT CORE DATE INTO HOTAB
212
INCREMENT CORPT
214
ROTATE ACTHO, DECREMENT DATE CNT
200
DATE CNT = 0 ?
202
RETURN
CHODT
186
TEMP = DAY, CORE MONTH
188
190
COMPARE TEMP WITH HOTAB
192
TEMP IN HOTAB ?
194
INSERT TEMP INTO HOTAB
196
INCREMENT RYRPT RDTPT HILPT CORPT
198

Fig. 9.


CHODT
YR = Y10' Y10°
220
GET Y10³ Y10²
222
Y10³ Y10² = 20 ?
224
N
Y
YR = YR + 100
226
#LEAP YR = (YR-REFYR)/4
228
#LEAP YR = (3*#LEAP YR)+MOD 4(YR-REF YR)
230
# DEC = MOD 7 [#LEAP YR+(2*# LEAP YR)]
232
TEMP DAY = (REFDT+7) - # DEC
234
TEMP DAY > HILIM ?
236
Y
N
DAY = TEMP DAY-7
240
DAY = TEMP DAY
238
RETURN
Fig. 10.

PROGRAMMABLE ELECTRONIC REAL-TIME LOAD CONTROLLER, AND APPARATUS THEREFOR, PROVIDING FOR UPDATING OF PRESET CALENDAR EVENTS

BACKGROUND OF THE INVENTION

This invention generally relates to programmable electronic real-time load controllers and apparatus therefor, and more particularly to such a controller and apparatus providing for the determination and storage of the actual day of a preset calendar event.

FIELD OF THE INVENTION

Programmable electronic real-time load controllers are known to the art for controlling the energization of a plurality of electrical loads in accordance with a predetermined time schedule. An example of such a controller can be found in U.S. Pat. No. 4,293,915, Carpenter et al., which is assigned to the assignee of the present invention. The controller in Carpenter et al. includes: a plurality of load control circuits, each load control circuit being adapted to be interconnected with an electrical load circuit or "load", and having a load-on state when its load is to be on, and a load-off state when its load is to be off; a clock for accumulating real-time information; and, a data processor, operating under control of a stored program, for responding to real-time information obtained from the clock to effect control of the load-on and the load-off states of each of the plurality of load control circuits in accordance with a time schedule and other control information that has been stored in the data processor. The user of this controller may preprogram the time schedule by selecting a number of control events and associated event times for each of the plurality of loads. The control events and event times for each load can be assigned to each day of the week and stored in a corresponding day schedule, and can be assigned to a holiday and stored in a corresponding holiday schedule. Normally, the control events and event times in each day schedule are utilized upon occurrence of the corresponding day in real-time; however, the control events and event times in the holiday schedule are utilized upon the occurrence of a preset holiday data in real-time. Each selected control event either causes the load to be turned on, to be turned off, or to be duty-cycled, from a time in real-time corresponding to the associated event time to a time in real-time corresponding to the event time of a subsequent control event for the load.

The clock specifically disclosed in Carpenter et al. is a weekly or seven-day clock. As a result, the preset holiday data for each load must be selected each week by the user. In addition, the transition from standard to daylight time, and the transition from daylight to standard time, both of which require modification to the real-time information in the clock, must be entered into the data processor by the user at the occurrence of those calendar events. An improved controller of the type specifically disclosed in Carpenter et al. includes a yearly or 365-day clock. This improved controller accordingly permits the user to preset a number of calendar events, such as the standard/daylight transition, the daylight/standard transition, and a number of holiday dates, by entering into the data processor the month and day of each calendar event. In the improved controller, the data processor adjusts the real-time information in the clock upon the occurrence in real-time of the month and day of the standard/daylight and daylight/standard transitions, and selects the holiday schedule for a load upon the occurrence in real-time of the month and day of each holiday date.

A disadvantage of this improved controller is that the majority of calendar events, being entered as they are by month and day, are valid only for a single year. Although the day of certain holidays such as Christmas remains the same from year to year, the days of certain other holidays such as Labor Day and the days of the standard/daylight and daylight/standard transitions vary from year to year. For example, Labor Day is always the first Monday in September, the standard/-daylight transition is typically the last Sunday in April, and the daylight/standard transition is typically the last Sunday in October. Accordingly, certain of the calendar events must be reentered eacy year in order for those calendar events to valid during the coming year. The present invention is therefore directed in its preferred form to a controller of the type described, and an apparatus therefor, that provide for the periodic, e.g., yearly, determination and storage of the actual day of a preset calendar event.

SUMMARY OF THE INVENTION

The invention consists of an apparatus for determining and storing the current numeric day of a preset calendar event of the type that always occurs on the same alphabetic day always having the same relationship to the beginning or end of the same month in any year but whose numeric day varies from year to year. The apparatus comprises: means storing a reference day value representing the numeric day of the calendar event in a given year; means storing a reference year value representing the numeric year of that given year; means determining a real-time year value representing the numeric year of the real-time year; means determining a current numeric day value for the calendar event from the reference day value, the reference year value, and the real-time year value; and, means storing the current numeric day value.

Preferably, the means determining the current numeric day value includes: means determining a temporary numeric day value for the calendar event by decrementing the reference day value in relation to the number of leap years and nonleap years that have elapsed between the year represented by the reference year value and the year represented by the real-time year value; means determining a numerical limit for the current numeric day value in view of the relationship of the corresponding alphabetic day to the beginning or end of the month in which the alphabetic day occurs; means comparing the temporary numeric day value with the numerical limit and adjusting the temporary numeric day value so that the temporary numeric day value falls within the numerical limit; and, means selecting the temporary numeric day value as the current numeric day value.

In its preferred form, this apparatus is implemented as an improvement to an electronic controller that includes at least one load control circuit for controlling the energization state of a corresponding electrical load, and, a data processor operating under control of a stored program. The data processor accumulates real-time information representing the numeric day, month and year of real-time, stores a predetermined schedule for control of the load, compares its predetermined schedule for load control with its real-time information, and causes the load control circuit to control the energization state of the load in accordance with that comparison. The data processor also stores at least one calendar event of the type described by its numeric day and its numeric month, compares the stored numeric day and numeric month of the calendar event with its real-time information, and undertakes a predetermined control action relating either to its real-time information or to its schedule for load control upon the occurrence in real-time of the stored numeric day and numeric month of the calendar event.

In the improvement, such a data processor is operative to:

select, as a reference day, the numeric day of the calendar event in a given year;

select, as a reference year, the numeric year of that year; and,

update the stored numeric day of the calendar event by periodically:

(a) determining the real-time numeric year from its real-time information;

(b) determining a temporary numeric day for the calendar event by decrementing the reference day in relation to the number of leap years and nonleap years that have elapsed between the reference year and the real-time year;

(c) determining a numerical limit for the numerical day of the calendar event in view of the relationship of the corresponding alphabetic day to the beginning or end of the month in which the alphabetic day occurs;

(d) comparing the temporary numeric day with the numerical limit and adjusting the temporary numeric day so that the temporary numeric day falls within the numerical limit; and,

(e) storing the temporary numeric day as the numeric day of the calendar event for the real-time year.

Preferably, the stored numeric day of the calendar event is updated at yearly intervals, such as at the beginning of each real-time year.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention can best be understood by reference to the following portion of the specification, taken in conjunction with the accompanying drawings in which:

FIG. 1 is a block diagram illustrating a programmable electronic real-time load controller including a microprocessor having a program memory and a data memory;

FIG. 2 is a schematic representation of certain registers and bytes in the data memory, including those storing the preset calendar events used by the controller;

FIG. 3 is a schematic representation of certain tables in the program memory;

FIG. 4 is a flow chart of the main program steps undertaken by the microprocessor in periodically updating the preset calendar events;

FIG. 5 is a flow chart of the program steps undertaken by the microprocessor in a ULEAP routine;

FIG. 6 is a flow chart of the program steps undertaken by the microprocessor in a RPDST routine;

FIG. 7 is a flow chart of the program steps undertaken by the microprocessor in a DSTNX routine;

FIG. 8 is a flow chart of the program steps undertaken by the microprocessor in a LOOK routine;

FIG. 9 is a flow chart of the program steps undertaken by the microprocessor in a RPHOL routine; and,

FIG. 10 is a flow chart of the program steps undertaken by the microprocessor in a CHODT routine.

DESCRIPTION OF A PREFERRED EMBODIMENT

Referring now to FIG. 1, the programmable electronic real-time load controller includes a microprocessor 20 that contains: a CPU 22; a program memory 24; and, a data memory 26. Microprocessor 20 receives input and control data from time and load programming controls 28 and also receives time-base information from a hardware clock 30, and outputs display data to a display 32 and control signals to a plurality of load control circuits 34, one for each load.

Data memory 26 contains: a real-time clock whose contents are periodically updated by the time-base information from clock 30; a predetermined time schedule for load control; and, certain other control data relating to that time schedule. Through time and load programming controls 28, a user may enter or alter the information in the real-time clock, may enter or alter the time schedule for load control, and may cause certain data to be displayed by display 32. In the time schedule, each load has assigned thereto a holiday schedule and a plurality of day schedules, one for each day of the week. Each day schedule and each holiday schedule may include one or more control events and associated event times, with each control event representing a predetermined control function for the associated load that is to begin at the associated event time in real-time.

By comparing the information in the real-time clock with the event times in the time schedule, microprocessor 20 determines the occurrence in real-time of each control event for a load and implements the control function represented by that control event by transmitting appropriate control signals to the corresponding one of the plurality of load control circuits 34. Normally, the microprocessor looks at the day schedule for the load corresponding to the real-time day of the week; however, the microprocessor looks at the holiday schedule for the load upon the real-time occurrence of a preset holiday data. For further details concerning the structure and operation of a controller of this type, reference should be made to the Carpenter et al. patent previously discussed.

Referring additionally now to FIG. 2, the real-time clock in data memory 26 consists of a RT register that is subdivided into a plurality of fields. The fields include: MINUTE fields containing the tens ($M10^1$) and units ($M10^0$) of the real-time numeric minute; HOUR fields containing the tens ($H10^1$) and units ($H10^0$) of the real-time numeric hour; and AM/PM field indicating whether the minute and hour are am or pm; DAY fields containing the tens ($D10^1$) and units ($D10^0$) of the real-time numeric day; MONTH fields containing the tens ($MO10^1$) and units ($MO10^0$) of the real-time numeric month; and, YEAR fields containing the thousands ($Y10^3$), the hundreds ($Y10^2$), the tens ($Y10^1$) and the units ($Y10^0$) of the real-time numeric year. The RT register accordingly comprises a yearly or 365-day clock, and the information contained therein is periodically updated by CPU 22 using a routine that references the time-base information in hardware clock 30.

Referring now back to FIG. 1, a plurality of controls 36 are provided that permit the user to enter or alter certain calendar events in data memory 26 to be used by microprocessor 20 in effecting load control. Controls 36 include a MONTH control and a DAY control that are used to enter or alter the numeric month and numeric day of a calendar event, a DST control that is used to indicate that the calendar event selected by the MONTH and DAY controls is the standard/daylight transition, a STD control that is used to indicate that the calendar event selected by the MONTH and DAY controls is the daylight/standard transition, a HOLIDAY control that is used to indicate that the calendar event selected by the MONTH and DAY controls is a holiday, a EOM control that is used to indicate that the standard/daylight transition or the daylight/standard transition is referenced to the end of the month, and a BOM control that is used to indicate that the standard/daylight transition or the daylight/standard transition is referenced to the beginning of the month.

In the situation where the selected calendar event is the standard/daylight transition, the month and day thereof are stored in a DST register in data memory 26. Referring again to FIG. 2, the DST register consists of: MONTH fields containing the tens ($MO10^1$) and units ($MO10^0$) of the numeric month; and, DAY fields containing the tens ($D10^1$) and units ($D10^0$) of the numeric day. In the situation where the selected calendar event is the daylight/standard transition, the month and day thereof are stored in a STD register in data memory 26 that consists of: MONTH fields containing the tens ($MO10^1$) and units ($MO10^0$) of the numeric month; and, DAY fields containing the tens ($D10^1$) and units ($D10^0$) of the numeric day. In the situation where the selected calendar event is a holiday, the month and day thereof are stored in a HOTAB register in data memory 26 that includes a plurality n of HOLIDAY fields in which are stored the numeric month and numeric day of up to n holidays. The HOLIDAY fields are arranged in chronological order, and each HOLIDAY field consists of: MONTH fields containing the tens ($MO10^1$) and units ($MO10^0$) of the numeric month; and, DAY fields containing the tens ($D10^1$) and units ($D10^0$) of the numeric day.

Entry of data into the DST and STD registers and into the various HOLIDAY fields in the HOTAB register is restricted so that the numeric month and numeric day are prospective only, that is, they must be a month and day that are equal to or in advance of the real-time numeric month and numeric day. Upon occurrence in real-time of the numeric month and numeric day contained in the DST register, CPU 22 advances the real-time information in the RT register by one hour. Upon occurrence in real-time of the numeric month and the numeric day in the STD register, CPU 22 retards the real-time information in the RT register by one hour. Upon occurrence of the numeric month and the numeric day in any HOLIDAY field in the HOTAB register, CPU 22 looks at the holiday schedule for any load that has been selected therefor rather than the day schedule therefor and concurrently erases the corresponding HOLIDAY field in the HOTAB register. Routines for implementing the foregoing procedures will be readily apparent to those of skill in the art by reference to analogous routines discussed in the Carpenter et al. patent.

The essential task of the invention is to provide a means by which the numeric month and numeric day of the standard/daylight transition, of the daylight/standard transition, and of certain selected "core" holidays may be periodically and automatically updated and

reentered into the DST and STD registers and into corresponding HOLIDAY fields in the HOTAB register, in order that a user does not have to periodically redetermine and reenter the numeric month and numeric day of those calendar events.

In addressing this task, it must first be recognized that each calendar year can be visualized as consisting of a plurality of successive monthly matrices, the monthly matrices being ordered by numeric month or by alphabetic month. Each monthly matrix consists of a predetermined plurality of numeric days, the numeric days being arranged in columns by alphabetic days and in rows by numeric weeks. Certain calendar events always occur each year on the same numeric day in the same numeric week in the same numeric month. The position of each calendar event of this type in the corresponding monthly matrix accordingly will shift from year to year. Examples of calendar events of this type are the holidays Christmas (12/25), Veteran's Day (11/11), Independence Day (7/4), and New Year's (1/1). Certain other calendar events always occur each year on the same alphabetic day in the same numeric week in the same numeric month. Although the position of each calendar event of this type remains the same in the corresponding monthly matrix from year to year, the numeric day shifts from year to year. Examples of calendar events of this type are: the standard/daylight transition (which typically occurs on the last Sunday in April); the daylight/standard transition (which typically occurs on the last Sunday in October); Thanksgiving (which always occurs on the fourth Thursday in November); Labor Day (which always occurs on the first Monday in September); Memorial Day (which always occurs on the last Monday in May); and, Presidents' Day (which always occurs on the third Monday in February).

An investigation of the calendar reveals:

for a numeric day in a current year that is equal to or greater than March 1 and is equal to or less than December 31
  if the next year is not a leap year, the numeric day for each alphabetic day will be decreased by one the next year (1)

for a numeric day in a current year that is equal to or greater than January 1 and that is equal to or less than February 28
  if the the current year is not a leap year, the numeric day for each alphabetic day will be decreased by one the next year (2)

for a numeric day in a current year equal to or greater than March 1 and equal to or less than December 31
  if the next year is a leap year, the numeric day for each alphabetic day will be decreased by two the next year (3)

for a numeric day in a current year equal to or greater than January 1 and equal to or less than February 29
  if the current year is a leap year, the numeric day for each alphabetic day will be decreased by two the next year. (4)

Further recognizing that a specific calendar event such as Labor Day occured on a numeric "reference day" (e.g., 01) in a numeric "reference year" (e.g., 1980), the number of leap years and nonleap years that will elapse to any future year can be determined. From this inforation, the number of "decrement days" that the reference day must decremented by to give the numeric day for the specific calendar event in any future year can then be determined.

From statements (1) through (4), it can be appreciated that the reference year should be chosen as follows. If the numeric day of the calendar event is equal to or greater than March 1 and is equal to or less than December 31, the reference year should be a leap year. If the numeric day of the specific calendar event is equal to or greater than January 1 and is equal to or less than February 28 or February 29, the reference year should be the first year following a leap year.

After determining the decrement days for any specific calendar event, a further check must be made to determine as to whether the number of decrement days will move the numeric day of the calendar event into the preceding week. In making this determination, the number of decrement days may be divided by seven. The resultant quotient is discarded and the resultant remainder is retained. The remainder is then investigated to determine if the remainder would decrement the reference day into the preceding week. If this determination is affirmative, the numeric day for the calendar event is the reference day minus the decrement days plus seven; if this determination is negative, the numeric day for the calendar event is the reference day minus the decrement days.

An easier procedure to determine the numeric day of the specific calendar event is to relate the reference day to the following week by adding seven to the reference day before it is decremented. The resultant "temporary" numeric day is then compared with the maximum numeric day or "high limit" that is possible for the numeric week of the calendar event. If this determination is affirmative, the numeric day is the temporary numeric day minus seven. If this determination is negative, the numeric day is the temporary numeric day. For those calendar events whose alphabetic days are referenced to the beginning of a month, the high limits for the numeric weeks are:

Week	High Limit Day
1	7
2	14
3	21
4	28

For those calendar events whose alphabetic days are referenced to the end of a month, the high limits are:

Week	High Limit Day
1	(Last day of month) - 28
2	(Last day of month) - 21
3	(Last day of month) - 14
4	(Last day of month) - 7
5	Last day of month

Exemplary routines that are executed by CPU 22 in periodically updating the calendar events in the aforesaid manner can be found in FIGS. 4 through 10, wherein CPU 22 references certain fixed data stored in program memory 24 as illustrated in FIG. 3 and certain variable data stored in data memory 26 as illustrated in FIG. 2.

Referring now to FIG. 4, the set of main program instructions illustrated therein are executed at an appropriate point in a main program loop through which CPU 22 repetitively passes. An example of such a main

program loop can be found in FIGS. 7(*a*) and 7(*b*) of the Carpenter et al. patent, and an appropriate point for insertion of the main program instructions in FIG. 4 would be in the REAL-TIME CLOCK routine illustrated in FIG. 8 of that patent.

Initially, CPU 22 enters step 100 in which the real-time information in register RT is updated as necessary by looking at the time-base information from oscillator 30. If the real-time month and day in the RT register correspond to the numeric month and day stored in the DST register, the real-time information in the RT register is advanced by one hour. If the real-time month and day correspond to the numeric month and day in the STD register, the real-time information in the RT register is retarded by one hour. Preferably, advancement and retardation of the real-time information are done at a specified time in real-time such as 2:00 a.m. From step 100, CPU 22 proceeds in step 102 to determine if real-time is 12:00 a.m. on January 1. If the determination in step 102 is negative, CPU 22 returns to its main program loop. If the determination in step 102 is affirmative, CPU 22 proceeds through routines ULEAP, RPDST, and RPHOL in successive steps 104, 106, and 108, and then returns to its main program loop.

During the ULEAP routine, CPU 22 determines the relation of the real-time year to a leap year, which information is used in the succeeding routines. During the RPDST routine, CPU 22 determines the numeric day of the next daylight/standard transition and stores that numeric day and the numeric month therefor in the DST register, and determines the numeric day of the next daylight/standard transition and stores that numeric day and the numeric month therefor in the STD register. During the RPHOL routine, CPU 22 determines the numeric day for each of the core holidays having a fixed alphabetic day and stores that numeric day and the corresponding numeric month in a corresponding HOLIDAY field in the HOTAB register, along with the numeric days and months of the core holidays having fixed numeric days. The information in the DST and STD registers and the information in the HOLIDAY fields in the HOTAB register insofar as the core holidays are concerned is thus periodically updated at yearly intervals.

Referring now to the ULEAP routine in FIG. 5, CPU 22 first gets the tens ($Y10^1$) and the units ($Y10^0$) of the real-time year stored in the YEAR fields in the RT register, in step 110, and then gets the thousands ($Y10^3$) and the hundreds ($Y10^2$) of the real-time year stored in the YEAR fields in the RT register, in step 112. In step 114, CPU 22 determines if the current real-time year is between 2000 and 2100, by determining if the thousands and hundreds of the current real-time year are equal to "20". If the determination in step 114 is affirmative, CPU 22, in step 116, adds "100" to the tens and units of the real-time year so as to distinguish the years occurring at or after the year 2000 from those occurring before the year 2000.

For the purpose of determining the standard/daylight transitions and the daylight/standard transitions, the reference year is chosen to be the leap year 1980. In step 118, CPU 22 subtracts the tens and units of the reference year 1980 from the tens and units of the real-time year, divides the result by "4" and stores the remainder in its A register. CPU 22 then, in step 120, stores the contents of the A register, plus "1", in its B register, and stores a mask "00010000" in the A register. The information stored in the A and B registers is in binary, eight-bit form. Taking the real-time year 1983 as an example, the A register will contain "00010000" as previously described and the B register will contain "00000100". From step **120**, CPU **22** in step **122** right rotates the contents of the A register by one bit position and decrements the number within the B register by one. CPU **22** then determines, in step **124**, if the B register contains "0". If the determination in step **124** is negative, CPU **22** returns to step **122** and continues to loop through steps **122** and **124** until the determination in step **124** is affirmative, whereupon CPU **22** in step **126** stores the contents of the A register in a LEAP byte. CPU **22** then returns to its main program instructions illustrated in FIG. 4 (and thereafter to the RPDST routine).

Reference should be made to the following Table I for a specific example of how CPU **22** proceeds through steps **118** through **126** in determining the information to be stored in LEAP.

TABLE I

	Real-Time Year = 1983	Reference Year = 1980	
(118)	A = MOD 4 (83–80)	=	00000011
(120)	B = A + 1	=	00000100
	A	=	00010000
(122)	RRA	A =	00001000
	DECB	B =	00000011
(124)		B ≠	0
(122)	RRA	A =	00000100
	DECB	B =	00000010
(124)		B ≠	0
(122)	RRA	A =	00000010
	DECB	B =	00000001
(124)		B ≠	0
(122)	RRA	A =	00000001
	DECB	B =	00000000
(124)		B =	0
(126)		LEAP =	00000001

From the foregoing, it can be appreciated that LEAP will contain information that uniquely relates not only the real-time year but also the immediately adjacent years to a leap year, as summarized in Table II.

TABLE II

Real-Time Year	LEAP
Leap year	00001000
First year following leap year	00000100
Second year following leap year	00000010
Third year following leap year	00000001

The information in LEAP is used by CPU **22** in determining the numeric day of certain calendar events, that is, the standard/daylight transition and the daylight/standard transition, as described hereinafter.

Referring now to FIG. 6, CPU **22** next enters the RPDST routine and executes successive steps **130, 132, 134**, and **136** in which the numeric day of the next standard/daylight transition is determined, that numeric day and the corresponding numeric month are stored in the DST register, the numeric day of the next daylight/standard transition is determined, and that numeric day and the corresponding numeric month are stored in the STD register. In determining the numeric days in steps **130** and **134**, CPU **22** uses a common DSTNX routine illustrated in FIG. 7.

Initially, CPU **22** gets the numeric month and the numeric day stored in the DST or STD register corresponding to the calendar event being updated. CPU **22** then determines, in step **142**, if the calendar event is referenced to the beginning of a month. In doing so, CPU **22** determines whether the EOM control or the BOM control (FIG. 1) has been actuated. Depending upon the results of this determination, CPU **22** either proceeds through a branch including steps **144, 146, 148**, and **150**, or through a branch including steps **152** and **154**, to detemine the high limit HILIM representing the maximum numeric day for the specific numeric week of the calendar event. At present, both the standard/daylight transition and the daylight/standard transition are referenced to the end of the month, whereupon the determination in step **142** is negative. As a result, CPU **22** proceeds in step **144** to find the last day of the month (LDOM) for the year preceding the real-time year, by reference to a LOOK routine set forth in FIG. 8.

Referring back to FIG. 3, program memory **24** includes a LDOM table in which are stored the numeric last days of each month of a calendar year, including the numeric last day of February for both leap and nonleap years. The entries in the LDOM table are stored in the order of increasing numeric months, with the exception that the numeric last day for February for a nonleap year is the first entry in the table. The entries in the LDOM table are pointed to by a LDOMPT pointer.

Referring now to the LOOK routine in FIG. 8, CPU **22** initially in step **156** sets LDOMPT to point to the first entry in the LDOM table, e.g., that containing the last numeric day for a nonleap year February. In step **158**, CPU **22** determines if the numeric month of the calendar event is February. If the determination in step **158** is negative, CPU **22** then proceeds in step **160** to move LDOMPT by the number of the numeric month for the calendar event. Due to the ordering of the LDOM table, LDOMPT thus points at the entry corresponding to the numeric month of the calendar event, whereupon CPU **22** in step **162** gets that entry and denominates it as the last day of the month for the year under investigation. It should be noted that the last day of the month is always the same for every month for every calendar year, excepting the month of February. Accordingly, CPU **22** normally will proceed through the LOOK routine in the manner described and will always do so in case of the present standard/daylight and daylight/standard transitions which occur in months other than February. Assuming, however, that the numeric month of the calendar event is February, the determination in step **158** is affirmative, whereupon CPU **22** determines in step **164** if the year under investigation is a leap year. While proceeding through step **164**, CPU **22** makes its determination by comparing the LEAP byte determined in step **126** in the UNLEAP routine with a table such as Table II. If the determination in step **164** is affirmative, CPU **22** proceeds through steps **160** and **162** as previously described. If the determination in step **164** is negative, CPU **22** bypasses step **160** and proceeds directly to step **162**, inasmuch as LDOMPT is pointing to the first entry in the LDOM table representing the last day of the month for a nonleap year February.

Returning now to FIG. 7, CPU **22** proceeds in step **146** to set a BIAS byte equal to the last day of the month that was found in step **144** minus the numeric day of the calendar event that was obtained in step **140**, divided by "7". CPU **22** then returns, in step **148**, to the LOOK routine and finds the last day of the numeric month of the calendar event for the real-time year. In step **150**, CPU **12** sets a HILIM byte, representing the maximum numeric day for the numeric week of the calendar event, equal to the last day of the month found in step 148, minus the product of BIAS and "7".

Assuming now that the calendar event is referenced to the beginning of the month, the determination in step 142 is affirmative whereupon CPU 22, in step 152, subtracts "1" from the numeric day of the calendar event, divides the result by "7", adds "1", and stores the result in BIAS. In step 154, CPU 22 then sets HILIM equal to the product of BIAS, as determined in step 152, and "7".

Once having determined the high limit, CPU 22 then proceeds to determine the numeric day for the calendar event for the real-time year. In step 164, CPU 22 sets the numeric day obtained in step 140 to the numeric day plus "7". CPU 22 then determines, in step 166, if the numeric month obtained in step 140 is less than "3". If the determination in step 166 is affirmative, CPU 22 next determines in step 168 if the preceding year was a leap year, again by referring to the LEAP byte and a table such as Table II. If the determination in step 168 is negative, CPU 22 decrements the numeric day by "1" in step 170. If the determination in step 168 is affirmative, CPU 22 decrements the numeric day by "2" in step 172. Assuming that the numeric month is equal to or greater than "3", the determination in step 166 is negative whereupon CPU 22 proceeds in step 174 to determine if the real-time year is a leap year, again by referring to the LEAP byte and a table such as Table II. If the determination in step 174 is negative, CPU 22 proceeds to step 170 as previously described. If the determination in step 174 is affirmative, CPU 22 proceeds to step 172 as previously described. The various actions undertaken by CPU 22 in steps 166 through 174 can best be understood by reference to statements (1) through (4) previously described.

From either step 170 or step 172, CPU proceeds in step 176 to determine if the numeric day is greater than the HILIM byte determined in either of steps 150 or 154. If the determination in step 176 is affirmative, the numeric day has been moved into the succeeding week, whereupon CPU 22 proceeds in step 178 to subtract "7" from the numeric day. From step 178 or from a negative determination in step 176, CPU 22 then returns to the RPDST routine.

The operation of CPU 22 while passing through the DSTNX routine will be further explained with reference to the specific example in Table III.

TABLE III

	Real-time year =	1983
	Last standard/daylight transition =	04/25/82
	LEAP =	00000001
(144)	LDOM =	30
(146)	BIAS =	(30 − 25)/7
	=	0
(148)	LDOM =	30
(150)	HILIM =	30 − (7 * 0)
	=	30
(164)	DAY =	25 + 7
	=	32
(166)	MONTH =	4 > 3
(174)	LEAP ≠	00001000
(170)	DAY =	32 − 1
	=	31
(176)		31 > 30
(178)	DAY =	31 − 7
	=	24

When CPU 22 has passed through the DSTNX routine in step 130, the numeric month obtained in step 140 and the numeric day obtained in either steps 176 or 178

are stored in step 132 in the appropriate fields in the DST register and accordingly represent the month and day of the next standard/daylight transition. When CPU 22 has proceeded through the DSTNX routine in step 134, the numeric month obtained in step 140 and the numeric day obtained in either steps 176 or 178 are stored in step 136 in the appropriate fields in the STD register and accordingly represent the month and day of the next daylight/standard transition.

After completing the RPDST routine, CPU 22 then enters the RPHOL routine as previously described. Referring now to FIG. 3, program memory 24 includes a CORE table containing certain information concerning the plurality of core holidays whose numeric month and numeric day will be automatically reentered into the HOTAB table as CPU 22 passes through the RPHOL routine, provided that each core holiday has been selected as a holiday by the user. The core holidays are divided into two groups: a first group consisting of those core holidays whose numeric day changes from year to year; and, a second group consisting of those core holidays whose numeric day remains the same from year to year. The first group includes Presidents' Day, Memorial Day, Labor Day, and Thanksgiving, and the second group includes New Year's, Independence Day, Veteran's Day, and Christmas. For those core holidays in the first group, the CORe table stores the numeric month of the holiday. For those core holidays in the second group, the CORE table stores the numeric month and numeric day of the holiday. The table entries are arranged in the CORE table in chronological order within each group and are pointed to by a CORPT pointer. Also stored in program memory 24 are a REFYR table, a REFDT table, and a HILIM table. The REFYR table contains the tens and units of the reference year for each core holiday in the first group, and the entries in the REFYR table are pointed to by a RYRPT pointer. The REFDT table contains the reference day for each core holiday in the first group, and the entries in the REFDT table are pointed to by a RDTPT pointer. The HILIM table contains the high limit day for each core holiday in the first group, and the entries in the HILIM table are pointed to by a HILPT pointer. The various entries in the REFYR, REFDT, and HILIM tables are arranged in chronological order. The selection or deselection of any core holiday as a holiday for the controller is signified by the setting or clearing of a corresponding bit in an ACTHO byte in data memory 26, as illustrated in FIG. 2.

Referring now to the RPHOL routine in FIG. 9, CPU 22 in step 180 initializes the RYRPT, RDTPT, HILPT, and CORPT pointers to the initial entry in each of the REFYR, REFDT, HILIM, and CORE tables, whereupon those pointers point to the entries associated with the first-in-time core holiday in the first group (for which a numeric day determination must be made). CPU 22 also obtains the first bit (B0) of the ACTHO byte (which likewise indicates whether or not the first-in-time core holiday in the first group has been selected as a holiday), and sets a DATE CNT counter to "8". In step 182, CPU 22 determines if DATE CNT is less than "5". If the determination in step 182 is negative, the core holiday to be investigated is one of those in the first group. If the determination in step 182 is affirmative, the core holiday to be investigated is one of the core holidays in the second group (for which no numeric day determination need be made). Assuming that the determination in step 182 is negative, CPU 22 proceeds in step 184 to see if the bit of ACTHO that has been obtained (e.g., in step 180) is set. If the pointed-to core holiday has been selected as a holiday, the determination in step 184 is affirmative whereupon CPU 22 proceeds in step 186 to a CHODT routine illustrated in FIG. 10.

During the CHODT routine, the numeric day of the pointed-to core holiday is determined for the real-time year. In step 220, CPU 22 sets a YR byte equal to the tens and units of the real-time year contained in the YEAR fields in the RT register. In step 222, CPU 22 gets the thousands and hundreds of the real-time year from the appropriate YEAR fields in the RT register. Thereafter, CPU 22 determines, in step 224, if the thousands and hundreds of the real-time year equal "20", that is, if the real-time year is between the year 2000 and the year 2100. If the determination in step 224 is affirmative, CPU 22 proceeds in step 226 to add 100 to the YR byte so as to distinguish the years occurring before the year 2000 from those occurring at or after the year 2000. From either step 226 or from a negative determination in step 224, CPU 22 then determines, in step 228, the number of leap years that have elapsed from the reference year of the core holiday to the real-time year. In doing so, CPU 22 subtracts the pointed-to reference year in the REFYR table from the real-time year in YR, divides the result by "4", and stores the quotient in a #LEAPYEAR byte. CPU 22 next, in step 230, determines the number of nonleap years that have elapsed from the reference year to the real-time year. In doing so, CPU 22 multiplies "3" by the year in #LEAPYEAR, adds thereto the remainder after dividing the difference between the years in YR and REFYR by "4", and stores the result in a #$\overline{\text{LEAPYEAR}}$ byte. Using #LEAPYEAR and #$\overline{\text{LEAPYEAR}}$, CPU 22 then determines in step 232 the number of decrement days by multiplying the year in #LEAPYEAR by "2", adding the result to the year in #$\overline{\text{LEAPYEAR}}$, dividing the result by "7", and storing the remainder in a #DEC byte. A temporary numeric day for the core holiday is then determined in step 234 by adding "7" to the pointed-to reference day in the REFDT table, subtracting the decrement days in #DEC therefrom, and storing the result in a TEMP DAY byte. In step 236, CPU 22 determines if the temporary numeric day in TEMP DAY is greater than the pointed-to high limit in the HILIM table. If the determination in step 236 is negative, CPU 22 in step 238 selects the temporary numeric day as the numeric day for the core holiday for the real-time year. If the determination in 236 is affirmative, CPU 22 in step 240 selects the temporary numeric day minut "7" as the numeric day. From either step 238 or step 240, CPU 22 returns to the RPHOL routine.

Those skilled in the art will appreciate that the steps in the CHODT routine are equivalent to those in the ULEAP and RPDST routines, but that the determination of the numeric day is made in a slightly different manner in order to accommodate the differing reference years, reference days, and high limits for the various core holidays in the first group. The example found in Table IV will further illustrate the CHODT routine.

TABLE IV

	Real-time year	=	1983
	Core holiday	=	President's Day
	Reference year	=	1981
	Reference day	=	16
	High limit	=	21
(228)	#LEAPYR	=	(83 − 81)/4
		=	0
(230)	#LEAPYR	=	(3 * 0) + MOD 4 (83 − 81)
		=	0 + 2
		=	2
(232)	#DEC	=	MOD 7 [2 + (2 * 0)]
		=	2
(234)	TEMP DAY	=	(16 + 7) − 2
		=	21
(236)	TEMP DAY	=	HILIM
(238)	DAY	=	TEMP DAY
		=	21

Returning now to FIG. 9, CPU 22 in step 188 stores the numeric day determined in the CHODT routine and the pointed-to numeric month for the core holiday in the CORE table in a TEMP byte. CPU 22 then determines in step 190 if all HOLIDAY fields in the HOTAB table contains entries. If the determination in step 190 is negative, CPU 22 then in step 192 compares the numeric month and numeric day in TEMP with those in the various HOLIDAY fields in the HOTAB table, and determines in step 194 if a match has been found. If the determination in step 194 is negative, CPU 22 then in step 196 inserts the numeric month and numeric in TEMP in the appropriate chronological HOLIDAY field in the HOTAB table, rearranging, if necessary, the entries therein to assure that all entries are in chronological order.

If the pointed-to core holiday has not been selected as a holiday by the user, or if all of the HOLIDAY fields in the HOTAB register contain entries, or if the numeric month and numeric day in TEMP are already contained in the HOTAB register, the determination in step 184 is negative, or the determination in step 190 is affirmative, or the determination in step 194 is affirmative. In any of these situations, or from step 196, CPU 22 proceeds in step 198 to increment the RYRPT, RDTPT, HILPT, and CORPT pointers to the next entries in the corresponding REFYR, REFDT, HILIM, and CORE tables. In succeeding step 200, CPU 22 gets the next succeeding bit in the ACTHO byte and decrements the number in the DATE CNT counter. In step 202, CPU 22 determines if the number in DATE CNT is "0". Assuming that all core holidays have not yet been investigated, the determination in step 202 is negative, whereupon CPU 22 returns to step 182.

CPU 22 continues to loop through the RPHOL routine as described until all of the core holidays in the first group have been investigated. Following the investigation of the last core holiday in the first group, the determination in step 182 is affirmative whereupon CPU 22 executes steps 204, 206, 208, 210, 212, and 214 for each core holiday in the second group. The actions taken by CPU 22 in these steps are similar to those previously described for the core holidays in the first group, with the exception that the numeric day of the core holiday is not determined, but rather obtained from the appropriate entry in the CORE table. As a result, the numeric month and the numeric day of each core holiday in the second group is stored in the appropriate chronological HOLIDAY field in the HOTAB register, provided that the core holiday has been selected as a holiday, the HOTAB register is not full, and that the numeric month and numeric day of the core holiday are not already contained in the HOTAB register.

While the invention has been described by reference to a preferred embodiment and several examples, it is to be clearly understood by those skilled in the art that the invention is not limited thereto and that the scope of the invention is to be interpreted only in conjunction with the appended claims.

The embodiments of the invention in which an exclusive property or privilege is claimed are defined as follows:

1. In an electronic controller that includes at least one load control circuit for controlling the energization state of a corresponding electrical load, and, a data processor operating under control of a stored program for: accumulating real-time information representing the numeric day, month and year of real-time; storing a predetermined schedule for control of the load; comparing its predetermined schedule for load control with its real-time information; causing the load control circuit to control the energization state of the load in accordance with that comparison; storing at least one calendar event by its numeric day and its numeric month, the calendar event being of the type that always occurs on the same alphabetic day always having the same relationship to the beginning or end of the same month in any year; comparing the stored numeric day and numeric month of the calendar event with its real-time information; and, undertaking a predetermined control action relating either to its real-time information or to its schedule for load control upon the occurrence in real-time of the stored numeric day and numeric month of the calendar event, the improvement wherein the data processor is operative to:

select, as a reference day, the numeric day of the calendar event in a given year;

select, as a reference year, the numeric year of said given year; and,

update the stored numeric day of the calendar event by periodically:

(a) determining the real-time numeric year from its real-time information;

(b) determining a temporary numeric day for the calendar event by decrementing said reference day in relation to the number of leap years and nonleap years that have elapsed between said reference year and said real-time year;

(c) determining a numerical limit for the numeric day of the calendar event in view of the relationship of the corresponding alphabetic day to the beginning or end of the month in which the alphabetic day occurs;

(d) comparing said temporary numeric day with said numerical limit and adjusting said temporary numeric day so that said temporary numeric day falls within said numerical limit; and,

(e) storing said temporary numeric day as the numeric day of the calendar event for said real-time year.

2. The improvement of claim 1, wherein the data processor is operative to update the stored numeric day of the calendar event at yearly intervals.

3. The improvement of claim 2, wherein the data processor is operative to update the stored numeric day of the calendar event at the beginning of each real-time year.

4. The improvement of claim 1, wherein said numerical limit is the maximum number that the numeric day of a calendar event can have in any year, given its relationship to the beginning or to the end of the month,

wherein the number "seven" is first added to said reference day before said reference day is decremented, and wherein said temporary numeric day is adjusted by subtracting the number "seven" therefrom only if said temporary numeric day is greater than said numerical limit.

5. The improvement of claim 4, wherein the numeric day of the calendar event always occurs in the same week referenced to the beginning of the month, and wherein the data processor is operative to determine said numerical limit by obtaining a stored value representing the maximum numeric day for that week of the month in any year.

6. The improvement of claim 4, wherein the numeric day of the calendar event always occurs in the same week referenced to the end of the month, and wherein the data processor is operative to determine said numerical limit by obtaining a stored value representing the last day of the month for said real-time year and by subtracting therefrom the product of the number "seven" and the number by which that week is referenced to the end of the month.

7. The improvement of claim 1, wherein said data processor is operative to:

select the stored numeric day of the calendar event for the year preceding said real-time year as said reference day;

select the numeric year of said preceding year as said reference year;

determine said temporary numeric day by;

(a) determining whether each of said real-time year and said reference year is a leap year or a nonleap year;

(b) subtracting the number "two" from said reference day if the numeric month of the calendar event is less than the number "three" and if said reference year is a leap year;

(c) subtracting the number "two" from said reference day if the numeric month of a calendar event is equal to or greater than the number "three" and if said real-time year is a leap year;

(d) subtracting the number "one" from said reference day if the numeric month of the calendar event is less than the number "three" and if said reference year is a nonleap year; and,

(e) subtracting the number "one" from said reference day if the numeric month of the calendar event is equal to or greater than the number "three" and if said real-time is a nonleap year.

8. The improvement of claim 1, wherein said data processor is operative to determine said temporary numeric day by:

determining the number of leap years that have elapsed between said reference year and said real-time year;

determining the number of nonleap years that have elapsed between said reference year and said real-time year;

determining a number of decrement days by obtaining the remainder, after dividing by the number "seven", of the sum of said number of nonleap years plus the product of the number "two" and said number of leap years; and,

subtracting said number of decrement days from said reference day.

9. The improvement of claim 8, wherein said data processor is operative to determine said number of leap years by subtracting said reference year from said real-

time year, by dividing the result by the number "four", and by saving the resultant quotient.

10. The improvement of claim 8, wherein said data processor is operative to determine said number of non-leap years by multiplying said number of leap years by the number "three", and by adding to the result the remainder, after dividing by the number "four", of said real-time year minus said reference year.

11. The improvement of claim 8, wherein said data processor is operative to select as said reference year the numeric year of a leap year if the numeric month of the calendar event is equal to or greater than the number "three".

12. The improvement of claim 8, wherein said data processor is operative to select as said reference year the numeric year of the first year following a leap year if the numeric month of the calendar event is less than the number "three".

13. The improvement of claim 1, wherein said calendar event is a time transition from standard to daylight time or from daylight to standard time.

14. The improvement of claim 1, wherein said calendar event is a holiday.

15. The improvement of claim 1, for use with a data processor that stores and utilizes the numeric day and numeric month of each of a plurality of calendar events of the type described, wherein the data processor undertakes each of the operations recited in claim 1 for each of the calendar events.

16. An apparatus for determining and storing the current numeric day of a preset calendar event that always occurs on the same alphabetic day always having the same relationship to the beginning or end of the same month in any year, comprising:

means for storing a reference day value representing the numeric day of the calendar event in a given year;

means for storing a reference year value representing the numeric year of said given year;

means determining a real-time year value representing the numeric year of the real-time year;

means determining a current numeric day value for the calendar event from said reference day value, said reference year value, and said real-time year value; and,

means storing said current numeric day value.

17. The apparatus of claim 16, wherein said means determining said current numeric day value includes:

means determining a temporary numeric day value for the calendar event by decrementing said reference day value in relation to the number of leap years and nonleap years that have elapsed between the year represented by said reference year value and the year represented by said real-time year value;

means determining a numerical limit for the current numeric day value in view of the relationship of the corresponding alphabetic day to the beginning or end of the month in which the alphabetic day occurs;

means comparing said temporary numeric day value with said numerical limit and adjusting said temporary numeric day value so that said temporary numeric day value falls within said numerical limit; and,

means selecting said temporary numeric day value as said current numeric day value.

* * * * *

)01-05-01

Reexam

LEVIN & HAWES LLP
ATTORNEYS AT LAW INTELLECTUAL PROPERTY LAW

364 FOREST AVENUE SUITE 13
LAGUNA BEACH, CALIFORNIA 92651
TEL 949.497.7676
FAX 949.497.7679
www.lagunalaw.com

#11

Patent ◆ Trademark ◆ Copyright ◆
Trade Dress ◆ and Related Litigation

OIPE
JAN 0 5 2001
PATENT & TRADEMARK OFFICE

Via: Express Mail ET 051659772 US

January 5, 2001
Box: Non-Fee Amendment
Assistant Commissioner for Patents
Washington, DC 20231

Dear Commissioner:

RECEIVED
JAN 0 5 2001
REEXAM UNIT

Enclosed is a **Housekeeping Amendment** in the merged cases:

Reissue Application No.:	)	**Group Art Unit:** 2177
09/512,592	)	
United States Patent No.:	)	**Examiner:** Paul Kulik
5,806,063	)	
Issued: September 8, 1998	)	**Attorney Docket No.:**
Applicant:	)	2039-154
Dickens-Soeder2000,LLC	)	
Reexamination Proceeding:	)	
90/005,592	)	
Filed: December 21, 1999	)	
Reexamination Proceeding:	)	
90/005,628	)	
Filed: February 2, 2000	)	
Reexamination Proceeding:	)	
90/005,727	)	
Filed: May 16, 2000	)	

Attorney Docket No.: 2039-154

This Amendment consists of:
Housekeeping Amendment of 36 pages
New Claims 16-76
Information Disclosure Statement
USPTO Form PTO/SB/08A
Supplemental Information Disclosure Statement Submission
Certificate of Mailing
Certificate of Service By Mail
Return Receipt Postcard

The fee and certification requirements of 37 C.F.R. §1.97 have been waived pursuant to the DECISION, *SUA SPONTE*, TO MERGE REEXAMINATION AND REISSUE PROCEEDINGS, mailed November 6, 2000. The fee for examination of claims in excess of the original filing fee have been paid in the above referenced Reissue Application.

If you have any questions, please do not hesitate to contact me.

Regards,

[illegible]

William C. Cray
Registration No. 27627

WCC/ns
Enclosures

[illegible]
the DECISION, *SUA SPONTE*, TO MERGE [illegible]
PROCEEDINGS, mailed [illegible]
of the original filing [illegible]

[illegible]

[illegible] Salama

January 5, 2001

BOX: NON-FEE AMENDMENT
Assistant Commissioner for Patents
Washington, DC 20231

CERTIFICATE OF MAILING UNDER 37 CFR § 1.10

Re: **Housekeeping Amendment** in the merged cases:

Reissue Application No.:)
09/512,592)
United States Patent No.:)
5,806,063)
Issued: September 8, 1998)
Applicant:)
Dickens-Soeder2000,LLC)
Reexamination Proceeding:)
90/005,592)
Filed: December 21, 1999)
Reexamination Proceeding:)
90/005,628)
Filed: February 2, 2000)
Reexamination Proceeding:)
90/005,727)
Filed: May 16, 2000)

Group Art Unit: 2177

Examiner: Paul Kulik

Attorney Docket No.:
2039-154

RECEIVED
JAN 0 5 2001
REEXAM UNIT

Attorney Docket No.: 2039-154

Enclosed with this Certificate of Mailing is:
Housekeeping Amendment of 36 pages
New Claims 16-76
Information Disclosure Statement
USPTO Form PTO/SB/08A
Supplemental Information Disclosure Statement Submission
Certificate of Service By Mail
Return Receipt Postcard

The fee and certification requirements of 37 C.F.R. §1.97 have been waived pursuant to the DECISION, *SUA SPONTE*, TO MERGE REEXAMINATION AND REISSUE PROCEEDINGS, mailed November 6, 2000. The fee for examination of claims in excess of the original filing fee has been paid in the above referenced Reissue Application.

[illegible]
Nanees Salama

Protecting Patents From The Beginning: The Importance of Information Disclosure Statements During Patent Prosecution

*Todd L. Juneau**
*and Jill K. MacAlpine***

* Mr. Juneau is a partner at the intellectual property law firm of in Washington, D.C. and can be reached at tljuneau@nathlaw.com. The views expressed herein are solely those of the authors, and do not represent those of the firm or its clients.

** Dr. MacAlpine is a technical advisor at Nath & Associates and is currently a first year law student at The George Washington University Law School in Washington, D.C.

I. INTRODUCTION

Filing a comprehensive Information Disclosure Statement (IDS) is essential to ensure validity and enforceability of a U.S. patent. In its simplest form, an IDS is a list of prior art such as other patents or publications which are relevant to the invention claimed in a patent application. This list should be submitted to the U.S. Patent and Trademark Office (PTO) on a specific form along with copies of the prior art cited therein. The issue of disclosure, or lack of disclosure, of material prior art is the subject of an extensive history of litigation. Considering the fact that the statutes and regulations governing the disclosure of prior art on their face appear to be quite straightforward, the vast number of cases concerning this topic is somewhat surprising. In actuality, these rules and regulations are complicated and fluid. As in all areas of the law, statutes turn on the definition(s) of the term therein and the element(s) that these terms qualify. Uncertainty surrounds terms such as "material," "filed in a timely fashion," and "inequitable conduct." As such, there is a real need for comprehension of the statutory laws, the caselaw and the rules governing Information Disclosure Statements, especially in light of the very serious ramifications of non-disclosure.

This article discusses the importance of filing comprehensive Information Disclosure Statements during patent prosecution. It also examines the duties which are unique to patent law and which are imposed on all those substantively associated with the preparation and prosecution of a patent application. The requirements for filing an IDS such as the appropriate format to be used, the timeliness of submission and what an IDS must disclose are outlined. Finally, the serious consequences of both intentional and unintentional non-disclosure are discussed.

II. The Basis for Filing an IDS

The principal reason for filing an IDS during patent prosecution is to ensure that a patent is valid and enforceable. One defense to a claim of patent infringement is the counterclaim of patent invalidity. One study has shown that the probability that a patent will be held invalid based on uncited prior art was 40.8%, while another study reported that 66–80% of patents held invalid involve prior art which was not cited to the PTO.[1] Thus, an IDS should be filed and should disclose all material prior art in order to avoid patent invalidity or unenforceability based on uncited prior art.

There are two main issues which must be addressed in order to ensure patent validity and enforceability. First, information which has been "considered" by an Examiner of the Patent Office during the prosecution of an application directly translates into a patent which is better able to withstand challenge by an infringer due to a strong presumption of validity. Hence, all material prior art must be disclosed to an Examiner in order to enjoy this presumption of validity. Second, every person substantively associated with the filing and prosecution of a patent application has an affirmative duty under 37 C.F.R. § 1.56 to disclose information which is material to patentability to the Patent Office. This duty must be discharged and a failure to do so can result in an unenforceable patent.

A. *Presumption of Validity*

By law, patents are presumed to be valid during patent infringement proceedings and this presumption must be overcome by an infringer who asserts invalidity as a defense.[2] This presumption is strong when prior art was before and considered by the Patent Office and weak when it was not.[3] When an IDS is filed during prosecution and the information therein is "considered" by an Examiner, the resulting patent is subject to a much higher standard of review than that employed for a patent wherein the disclosure of prior art was minimal or lacking.

Further, the burden to overcome this presumption of patent validity requires not only evidence of invalidity, but clear and convincing evi-

1 *See* John R. Allison & Mark A. Lemley, *Empirical Evidence on the Validity of Litigated Patents*, 26 AIPLA Quarterly 185, 234 (1998); *see also In re* Portola Packaging Inc., 110 F.3d 786, 789, 42 U.S.P.Q.2d 1295, 1298 (Fed.Cir. 1997) (citing Patent Reexamination: Hearings on S.1679 before the Senate Committee on the Judiciary, 96th Cong. § 14 (1980) (testimony of Commissioner Sidney Diamond, referring to a 1974 study showing that 66–80% of the patents held invalid involved uncited prior art).

2 35 U.S.C. § 282.

3 *See* Bolkcom v. Carborundum Co., 523 F.2d 492, 498, 186 U.S.P.Q. 466, 471 (6th Cir. 1975).

dence or else the defense will not be considered by the court.[4] Even if an infringer provides clear and convincing evidence of invalidity, there is an additional burden of overcoming the deference given to the PTO by the courts. Because a qualified government agency, which includes one or more examiners who are assumed to have some expertise in interpreting references and to be familiar with the level of skill in the art, is presumed to have done its job properly, a very high level of deference is created.[5]

The Court of Appeals for patent infringement lawsuits ensures that federal courts actually provide this deference to the PTO by requiring that "any" fact findings made by the PTO are reviewed by the Court of Appeals under the Administrative Procedures Act (APA) standard.[6] This means that the PTO's basis for granting a patent will not be overturned unless the agency's findings of fact are determined to be: (i) arbitrary, capricious or an abuse of discretion, or (ii) unsupported by substantial evidence.[7] The Supreme Court recognized this seemingly arcane area of the law to be of such importance that it overruled decades of previous decisions, and held that the Court of Appeals for the Federal Circuit must apply this "court/agency" framework rather than the traditional "court/court" standard of reviewing for clear error.[8] Thus, review under the APA standard is a high hurdle to overcome for any challenger. To take advantage of these procedural and evidentiary rules, patent applicants should disclose all material prior art in an IDS prior to issuance of the patent.

B. *Duty to Disclose*

The second issue to be addressed in order to ensure patent validity and enforceability is that of the duty of disclosure. Under federal patent regulations, specifically title 37 C.F.R. § 1.56, every individual associated with the filing and prosecution of a patent application has a duty to disclose information which is material to patentability, and a duty of candor and good faith in dealing with the PTO.[9] Failure to discharge these duties can result in an unenforceable patent.

These duties apply to: (1) each inventor named in the application; (2) each attorney or agent who prepares or prosecutes the application:

4 *See* Hughes Aircraft Co v. United States, 717 F.2d 1351, 219 U.S.P.Q. 473 (C.C.P.A. 1983).
5 *See* American Hoist & Derrick Co. v. Sowa & Sons, Inc. 725 F.2d 1350, 220 U.S.P.Q. 763 (Fed. Cir. 1984), *cert. denied,* 463 U.S. 821; Markman v. Westview Instruments, Inc. 52 F.3d 967, 968 (Fed. Cir. 1995), *affirmed* 517 U.S. 370, 134 L. Ed. 2d 577, 116 S. Ct. 1384 (1996).
6 *See* Dickinson v. Zurko, 527 U.S. 150, 50 U.S.P.Q.2d (BNA) 1930 (1999).
7 *See* cases cited *supra* note 6.
8 *See* cases cited *supra* note 6.
9 37 C.F.R. § 1.56(a).

and (3) every other person who is substantively involved in the preparation or prosecution of the application and who is associated with the inventor, with the assignee or with anyone to whom there is an obligation to assign the application. Individuals other than the attorney, agent or inventor can disclose information to either the attorney, agent, or inventor.

These duties exist with respect to each pending claim until the claim is canceled, withdrawn from consideration, or the application is granted or abandoned.[10] This duty to disclose is only deemed to be discharged if all information known to be material to patentability of any claim issued in a patent was cited by, or submitted to, the Office in the manner prescribed by Sections 1.97(b)–(d) and 1.98 of the Code of Federal Regulations (C.F.R.).

III. The Appropriate Manner and Time to File an IDS

An IDS must be compiled and submitted to the Patent Office in compliance with a number of rules and regulations. These rules and regulations, set forth in both the Manual of Patent Examining Procedures (M.P.E.P.) and in title 37 of the C.F.R., are discussed below and concern the format in which the IDS is presented, the timeliness of submission and the requisite copies of prior art references disclosed. The goal of the applicant in complying with these rules and regulations is to get the prior art references "considered" by the Patent Examiner.

A. *General Requirements*

Sections 1.97(b)–(d) and 1.98 of the C.F.R. state that information disclosed in an IDS will be considered by an Examiner of the Patent Office when: 1) listed, preferably using the Patent's Office's Form PTO-1449; 2) filed separately from the filing of the specification; 3) a copy of each document is provided (unless it is a copending application); and 4) the IDS is filed in a timely fashion.[11]

1. *The Listing Requirement*

37 C.F.R. § 1.98(b) requires that a list of all patents, publications, or other information be submitted for consideration by the Patent Office. Therefore, unless the references have been cited by the Examiner on a form, such as Form PTO-892, technically they have not been considered and the Examiner should notify the applicant of such in the next Office action.

10 *See supra* note 11.
11 37 C.F.R. §§ 1.97(b)–(d) and 1.98.

This listing requirement also applies to abstracts of references. Abstracts may be submitted, but should be cited as such on the Form PTO-1449. For example, an abstract of an article by Holt *et al.* published in 1996 from Chemical Abstracts Service (CAS) would be cited as "Holt, Chemical Abstract, vol. 125:86501, 1996."

a. Exceptions to the listing requirement—reliance on information in response to Office actions

In certain situations, information may be "considered" by the P.T.O. even if it was not submitted in an IDS. The Examiner's note to M.P.E.P. Section 6.49 provides various examples of such disclosures of information (e.g.: 37 C.F.R. §§ 1.97 and 1.98). In particular, it states that evidentiary documents submitted when replying to an Office action, such as a rejection or other official communication from the PTO, may be relied upon by an applicant for example to show that an element recited in the claim is operative or that a term used in the claim has a recognized meaning in the art. These evidentiary documents may be in any form but are typically in the form of an affidavit, declaration, patent, or printed publication. To the extent that a document is submitted as evidence directed to an issue of patentability raised in an Office action, and the evidence is timely presented, the applicant need not satisfy the requirements of 37 C.F.R. §§ 1.97 and 1.98 in order to have the Examiner consider the information contained in the document. In other words, compliance with the information disclosure rules is not a threshold requirement to have information considered when submitted by an applicant to support an argument being made in a reply to an Office action.

If the information is submitted by an applicant to support an argument being made in a reply to an Office action, the record should reflect whether the evidence was considered, but listing on a form (e.g., PTO-892, PTO-1449, or PTO/SB/08A and 08B) and appropriate marking of the form by the Examiner is not required. For example, if the applicant submits and relies on three patents as evidence in reply to an Office action and also lists those patents on a Form PTO-1449 along with two journal articles, but does not file a statement under 37 C.F.R. 1.97(e) or the fee set forth in 37 C.F.R. 1.17(p), it would be appropriate for the Examiner to indicate that the teachings relied on by the applicant in the three patents have been considered, but to line through the citation of all five documents on the Form PTO-1449 and to inform applicant that the Information Disclosure Statement did not comply with 37 C.F.R. § 1.97(c). Thus, situations may occur where some prior art gets considered even though the entire IDS did not comply with the regulations.

2. Filed Separately from the Specification

Apart from the aforementioned situations, prior art must be disclosed in compliance with Sections 1.97(b)–(d) and 1.98. For example, references listed in the background section of the patent application are not considered to be properly disclosed. Although caselaw, regulations, and the M.P.E.P. do not entirely foreclose the disclosure of information material to patentability by any other method than an IDS, M.P.E.P. Section 6.49.06 does specifically state that the listing of references in the specification or the body of a patent application is not considered a proper disclosure statement.

3. Submission of Copies of Disclosed Prior Art References

In addition to the listing requirement, a copy of each reference, including U.S. Patents, must be provided to the Examiner. M.P.E.P. Section 6.49.07 states:

The information disclosure statement. . .[will] comply with 37 CFR 1.98(a)(2), which requires a legible copy of each U.S. and foreign patent; each publication or that portion which caused it to be listed; and all other information or that portion which caused it to be listed.

Copies of other copending U.S. patent applications need not be supplied according to M.P.E.P. § 6.49.06, as long as they are properly cited on a separate form. The Examiner should obtain access to that application file within the Patent Office.

4. Timely Submission

The phrase "filed in a timely fashion" means filed: 1) before the first Official action on the merits is mailed by the Patent Office, 2) within three months of receiving the first Official action on the merits from a foreign patent office, 3) within three months of any person having a duty to disclose becoming aware of the information, 4) after the first Official action on the merits along with a fee, or 5) after the Examiner has indicated he or she will grant the patent provided the applicant files a petition to the Examiner after the Notice of Allowance but before the patent issues along with a fee and statement of reasons why it was not submitted earlier.[12]

If the IDS is not timely filed, it will be placed in the application file with the non-complying information not being considered.[13] Therefore,

12 37 C.F.R. § 1.97(i).; M.P.E.P. Section 6.51.

13 *See* cases cited *supra* note 1.

despite identifying, collecting and submitting the information, if it is not timely submitted, the IDS afford no protection if the patent is challenged.

IV. Content of an IDS: What Must an IDS Disclose?

Under 37 C.F.R. § 1.56, any information that is deemed to be "material to patentability" must be disclosed in an IDS. The language of this statute was modified in 1992 to emphasize a duty of candor and good faith in dealing with the Patent Office. Although the latter is broader than a duty to disclose material information, this does not affect the nature of the information required to be disclosed in an IDS.

A. *Information "Material to Patentability"*

Information is not material unless it comes within the definition of 37 C.F.R. § 1.56(b)(1) or (2). The relevant portion of this statute states:

> (b) Under this section, information is material to patentability when it is not cumulative to information already of record or being made of record in the application, and
> (1) It establishes, by itself or in combination with other information, a *prima facie* case of unpatentability of a claim; or
> (2) It refutes, or is inconsistent with, a position the applicant takes in:
> (i) opposing an argument of unpatentability relied on by the Office, or
> (ii) asserting an argument of patentability.
> A *prima facie* case of unpatentability is established when the information compels a conclusion that a claim is unpatentable under the preponderance of evidence, burden-of-proof standard, giving each term in the claim its broadest reasonable construction consistent with the specification, and before any consideration is given to evidence which may be submitted in an attempt to establish a contrary conclusion of patentability.

The test for materiality is not whether the prior art affects the novelty or obviousness of the invention but rather what a reasonable examiner would consider important in deciding whether to allow the patent application to issue as a patent.[14] References do not need to be anticipatory, i.e. novelty-destroying, to be considered material to patentability. Thus, information is deemed material if a reasonable examiner would have considered it important to the patentability of a claim.[15]

In *Fox Industries, Inc. v. Structural Preservation Systems, Inc.*, Fox Industries, Inc. sued the defendant, Structural Preservation Systems, Inc., for patent infringement.[16] During the suit, the district court found that

14 *See supra* note 11.

15 *See* J.P. Stevens & Co. v. Lex Tex Ltd., 747 F.2d 1553, 1559,223 USPQ 1089,1092 (Fed. Cir. 1984).

16 *See* Fox Industries, Inc. v. Structural Preservation Systems, Inc. 922 F.2d 801, 17 U.S.P.Q.2d 1579 (Fed. Cir. 1990).

more than a year prior to filing a continuation application of the original application, Fox had published a sales brochure describing the invention in the original patent application. Under title 35 U.S.C. § 102 (b), disclosure of the invention more than one year before filing the patent application destroys the novelty of the invention. Although the attorney used this brochure as source material for drafting the claims of the continuation application, he did not disclose it in an IDS in any of the four later applications which ultimately led to the patent in question. During litigation, these facts were uncovered by the defendants and used in their defense. The court found that the brochure was more relevant than any other single reference cited during prosecution and refused to enforce any of the claims in the patent at issue. On appeal, the court affirmed and ruled that a withheld reference which anticipates a claim in a patent satisfies the most stringent standard of material.[17] The Court of Appeals determined that Fox had knowledge of material prior art, had knowingly failed to disclose this art to the PTO and had an intent to deceive.[18] As a result, the court refused to enforce any of the patent's claims because of Fox's inequitable conduct and therefore determined that it was unnecessary to consider whether Structural Preservation Systems, Inc. had infringed the patent.[19]

B. *Classes of Information Considered to be Material*

The issue of what is "material" is frequently misunderstood by patent practitioners and thought to be restricted to prior art patents and publications. However, specific examples of information that is also considered "material to patentability"[20] are: information concerning possible prior public uses, sales, offers to sell, derived knowledge, prior inventions by others, inventorship conflicts, and the like.[21] This also includes prior art cited in search reports of a foreign patent office in a counterpart application,[22] information from or relating to copending U.S. patent applications,[23] information from related litigation[24] and, in particular. evidence of possible prior public use or sales, questions of inventorship or

17 *See id.*

18 *See id.*

19 *See id.*

20 M.P.E.P. § 2001.04.

21 *See id.*

22 *See supra* note 11.

23 *See* Armour & Co. v. Swift & Co., 466 F.2d 767, 779, 175 U.S.P.Q. 70, 79 (7th Cir. 1972).

24 *See* Critikon, Inc. v. Becton Dickinson Vascular Access, Inc., 120 F.3d 1253, 1258, 1259, 43 U.S.P.Q.2d 1666, 1670–71 (Fed. Cir. 1997) (patent held unenforceable due to inequitable conduct based on patentee's failure to disclose a relevant reference and for failing to disclose ongoing litigation).

prior art, allegations of "fraud," "inequitable conduct," "violation of duty of disclosure" and any assertion made during litigation which is contradictory to assertions made to the Examiner during pleadings, admissions, discovery (interrogatories, depositions, etc.) and testimony.[25] Information relating to claims copied from a patent, for example during interference proceedings to determine who the first inventor of an invention was, is also considered material.[26] The duty of disclosure also applies to statements or experiments introduced or discussed in the specification of the patent application.[27] For example, a statement that an experiment "was run" or "was conducted," when in fact the actual experiment was not run or conducted, is a misrepresentation of facts. Paper examples should not be described using the past tense.[28] Misrepresentations can also occur when experiments, although actually conducted, are inaccurately reported in the specification, such as when an experiment is changed by omitting one or more chemical reagents.[29]

C. *Classes of Information not Considered Material*

The above being said, the question arises as to what is not considered material to patentability or required as part of the duty under 37 C.F.R. § 1.56. There are some types of information that do not need to be disclosed. For example, disclosure of private unpublished documents is not required.[30] Information to show favorability of a patent such as evidence of commercial success of the invention does not need to be provided. Similarly, disclosure of information concerning the level of skill in the art for purposes of determining obviousness is not required.[31] Copies of references already cited to the PTO in a previous "parent" patent application are not required to be submitted again in a later application. Lastly, there is no duty to submit information which is not material to the patentability of any existing claim,[32] to explain the relevance

25 *See* Environ Prods., Inc. v. Total Containment, Inc., 43 U.S.P.Q.2d 1288, 1291 (E.D. Pa. 1997).

26 Where claims are copied or substantially copied from a patent, 37 C.F.R. § 1.607(c) requires applicant shall, at the time he or she presents the claim(s), identify the patent and the numbers of the patent claims. Failure to comply with 37 CFR 1.607(c) may result in the issuance of a requirement for information as to why an identification of the source of the copied claims was not made. Clearly, the information required by 37 CFR 1.607(c) as to the source of copied claims is material information under 37 CFR 1.56 and failure to inform the PTO of such information may violate the duty of disclosure.

27 *See* Steierman v. Connelly, 192 U.S.P.Q. 433 (Bd. Pat. Int. 1975); 192 U.S.P.Q. 446 (Bd. Pat. Int. 1976).

28 *See id.*

29 *See id.*

30 *See* Environmental Designs, Ltd. v. Union Oil Co. of California, 713 F.2d 693, 698, 218 U.S.P.Q. 865, 870 (Fed. Cir. 1983).

31 *See* case cited *supra* note 27.

32 *See supra* note 11.

of English language references, nor to disclose references which are merely cumulative. Accordingly, it is of paramount important to pay close attention to the claims of the patent application and exactly how the invention has been set forth.

D. *'Cumulative' defined*

Caselaw defines the term 'cumulative' as information which has the same features as, or is not substantively different from, the information already before the Examiner.[33]

In *Rolls-Royce Ltd. v. GTE Valeron Corp.*, the Court found that a reference that had not been cited was merely cumulative to another reference which had already been cited in an IDS. In this case, both references were novelty-destroying and anticipated the claims but they were structurally different from each other. The Court found that the structural differences did not render the second reference more material than the first, previously cited reference and found the references to be cumulative even though they were not identical.[34]

In determining whether uncited prior art is more material than that already before the Examiner, similarities and differences between the prior art and the claims of the application should be considered. Also of relevance are any portion of the art which teach away from the claimed invention.[35] Although this is admittedly a difficult and subjective decision to be made by experienced patent practitioners, submitting everything in your possession in order to err on the side of caution is to improperly place the applicant's duty at the Examiner's doorstep.

E. *Burying References*

The question of citing "too many" references is a highly subjective area of patent law.[36] Caselaw has held in some circumstances that the references were not "considered" by the Examiner, even though the citations in the IDS were initialed by the Examiner as having been considered, simply because of the number of references cited in an IDS. This is, of course, a hindsight judgment made during litigation in an attempt to reconstruct events after the fact and to second guess the reasons that a patent was granted.

33 *See id.*
34 *See* Rolls-Royce Ltd. v. GTE Valeron Corp., 800 F.2d 1101, 231 U.S.P.Q. 185 (Fed. Cir. 1986).
35 *See* Bausch & Lomb, Inc. v. Barnes-Hind/Hydrocurve, Inc., 796 F.2d 443, 448–9 (Fed. Cir. 1986).
36 *See* Molins v. Textron, 48 F.3d 1172, 33 U.S.P.Q.2d (BNA) 1823 (Fed. Cir. 1995).
See Golden Valley Microwave Foods v. Weaver Popcorn Co., 837 F.Supp. 144, 24 U.S.P.Q.2d (BNA) 1801 (1992).

On the other hand, "patent fraud," officially known as inequitable conduct, may be claimed by an infringer. The infringer may rely on the fact that certain references appear to be 'buried' among other, less relevant references in an IDS in order to obtain a judgment that the patent is unenforceable due to equitable concerns, despite not being invalid. In one such case, *Haney v. Timesavers, Inc.*, one defense raised against the claim of infringement was the allegation that the plaintiffs had 'buried' the relevant references amongst a group of ninety-one other, mostly far less relevant references. The Court of Appeals ruled that the references in question were not buried, but rather that they were listed in accordance with the applicable regulations.[37]

One possible option to avoid an assertion of "burying" a reference might be to point out in a letter to the Examiner particularly pertinent references or references which should be reviewed first or which relate to particular claims. The M.P.E.P. suggests highlighting particularly significant references submitted in a long list of references.

Another option might be to break up a particular unwieldy IDS into smaller IDS's containing more manageable numbers of references. This will hopefully increase the chances that the Examiner may actually read your submission, but should also reduce the acrimony created when you show up with a box of paper to the Examiner's art unit.

Yet a third option may be, if your client's budget allows, the creation and maintenance of a customized searchable database of references. Although you may have to file your IDS the old-fashioned way, an Examiner should have no problem signing off on your Forms PTO-1449 if you have also submitted a customized CD-ROM or private webpage where they can quickly search and retrieve documents considered material by the inventor(s) and attorney(s).

Although roadmapping and compartmentalizing are above and beyond the actual duty required, it may provide for better relations with the Examiner and take away a distracting challenge from future opponents.

V. Remedies and Consequences of Filing an Incomplete IDS

Some may decide to take their chances rather than go to all the trouble to submit the information. This is not advisable, however, experience has shown that good litigators will discover any documents in your files which were not, but which should have been, submitted to the Patent Office in an IDS.

37 *See* Haney v. Timesavers, Inc., 900 F. Supp. 1378 (1995).

Should material prior art be discovered by anyone substantively involved in the preparation or prosecution of the application after the patent has been granted, there are two possible corrective measures: reissue and re-examination. Both of these are limited to unintentional non-disclosure.

If material prior art was intentionally withheld, it is quite likely that the art will be discovered during any future litigation. Upon a finding of intentional non-disclosure, claims (or counterclaims) of inequitable conduct can be raised, threatening a patent's validity as well as an attorney's practice. Inequitable conduct can also involve antitrust implications which entitle a successful challenger to treble damages.

A. *Corrective Measures*

1. Reissue Application

Reissue proceedings are available to correct unintentional errors which make the patent invalid or inoperative.[38] Thus, reissue proceedings cannot cure inequitable conduct committed during the prosecution of the original application.[39] A reissue application is filed by the original applicants or their representatives and surrenders the entire patent for a new examination. Once reissued, the patent, according to 35 U.S.C. § 252, will be viewed as if the original patent had been granted in the amended form provided by the reissue.

If submission of previously unconsidered prior art is the only concern and the art raises a substantial new question of patentability, then re-examination is probably the more appropriate method to correct the error. If the considered prior art does not raise a substantial new question of patentability, there is a good chance it may in fact be a cumulative reference and no action need be taken. However, to ensure validity. and remove any distracting challenges from an infringers arsenal, reissuing a patent is a good preparatory step prior to enforcement actions.

2. Re-examination Proceedings

Re-examination proceedings are limited solely to re-examination of prior art, and are not available to correct intentional errors.[40] 35 U.S.C. § 304 is the statute that governs re-examination proceedings. This statute requires an examiner to determine whether a substantial new question of

38 M.P.E.P., Chapter 14, Correction of Patents.

39 *See In re* Clark, 522 F.2d 623, 187 U.S.P.Q. 209 (CCPA) 1975.
See Hewlett Packard v. Bausch & Lomb Inc., 882 F.2d 1556, 11 U.S.P.Q.2d 1750 (Fed. Cir. 1989).

40 *See* Patlex Corp. v. Mossinghoff, 758 F.2d 594, 601, 225 U.S.P.Q.2d 243, 248 (Fed. Cir. 1985).

patentability is raised by a re-examination request. Only if a substantial new question of patentability is raised, can a patent be re-examined.

In *In re Recreative Technologies Corp.*, the court reviewed the legislative history of the statute and determined that it reflected a "serious concern that reexamination not create new opportunities for abusive tactics and burdensome procedures."[41] The requirement that "[n]o grounds of reexamination were to be permitted other than on new prior art and sections 102 and 103" was a well-considered balance of the arguments for and against reexamination.[42]

Re-examination is barred for questions of patentability that were decided in the original examination.[43] Therefore, if the references were considered, but were not submitted in an IDS, reexamination is not available. The court in *In re Recreative Technologies Corp.* held that a prior art reference that served as a basis of a rejection in the prosecution of the original patent application could not support a substantial new question of patentability that would permit the institution of a reexamination proceeding.[44]

In another case, *In re Portola Packaging,* the court held that prior art which was previously before the original examiner could not support a reexamination proceeding despite the fact that it was not the basis of a rejection in the original prosecution.[45] The court held that, as long as the reference was before the original examiner, it was to be considered "old art".[46] Thus, re-examination is available for the sole purpose of considering completely unconsidered prior art which raises a substantial new question of patentability.

It should be noted that the American Inventor's Protection Act, signed into law on November 29, 1999, provides for the possibility of substantial third party involvement during reexamination. Accordingly, this decision should be made carefully in a highly competitive environment.

B. Consequences

1. Inequitable Conduct

Since 1976, the laws regarding disclosure of prior art have undergone substantial changes. Rules 56 (C.F.R.§ 1.56) was adopted in 1977

41 *See* H.R. Rep. No. 96-1307, at 3 (1980), *reprinted in* 1980 U.S.C.A.A.N. 6460, 6462.

42 *See id.*

43 *See id.*

44 *See In re* Recreative Technologies Corp., 83 F.3d 1394, 1397, 38 U.S.P.Q.2d 1776, 1778 (Fed. Cir. 1996).

45 *See* case cited *supra* note 1.

46 *See id.*

and amended in 1982, 1983, 1984, and again in 1985. The amendments permitted these matters to be appealed to the Board of Patent Appeals and Interferences.[47] Additionally, the amended Rule 56 made applicants, associates, and attorneys partners with the PTO for consideration of prior art. In this partnership, members bear a "duty of disclosure," violation of which constitutes "inequitable conduct."[48]

Under the present Rule 56, if inequitable conduct is found, under an 'abuse of discretion' standard, patents are held invalid and/or unenforceable.[49] Thus, a court will not, out of fairness, use a fraudulently obtained document to hold someone liable.[50] The penalty for inequitable conduct is rejection of all claims under 35 U.S.C. §§ 131 and 132, not "striking" of an application as in the pre-1977 rule.[51]

Inequitable conduct is comprised of two elements: materiality and intent. Thus, the doctrine of inequitable conduct requires a two-step analysis. First, it must be determined whether the withheld references satisfy a threshold level of materiality and whether the applicant's conduct satisfies a threshold showing of intent to mislead. If and only if these two determinations conclude that the thresholds are satisfied, are materiality and intent balanced. The more material the omission, the less culpable the intent required, and vice versa.[52]

As previously discussed, prior art is deemed material if is likely that a reasonable examiner would consider it important in deciding whether or not to allow the application to issue as a patent, but materiality of an undisclosed reference does not presume an intent to deceive.[53] Further, a mere showing that information material to patentability was not disclosed does not establish inequitable conduct.[54]

An infringer asserting an inequitable conduct defense must prove by clear and convincing evidence that the applicant or his or her attorney failed to disclose material information or submitted false information to the PTO with an intent to deceive.[55] The infringer must, therefore, provide clear and convincing proof of: (1) prior art or information that is

47 *See In re* Harita, 847 F.2d 801, 6 U.S.P.Q.2d 1930 (1988).

48 *See* case cited *supra* note 48.

49 *See* Kingsdown Medical Consultants, Ltd. v. Hollister Inc., 863 F.2d 867, 876, 9 U.S.P.Q.2d 1384, 1392 (Fed. Cir. 1988).

50 *See* Halliburton Co. v. Schlumberger Technology Corp., 925 F.2d 1435, 17 U.S.P.Q.2d 1834 (Fed. Cir. 1991).

51 *See* case cited *supra* note 48.

52 *See* case cited *supra* note 51.

53 *See* Allen Organ Co. v. Kimball International, Inc. 839 F.2d 1556 (Fed. Cir. 1988).

54 *See* Litton Industrial Products, Inc. v. Solid State Systems Corp., 755 F.2d 158, 225 U.S.P.Q. 34 (Fed. Cir. 1985).

55 *See* case cited *supra* note 17.

material; (2) knowledge chargeable to the applicant of that prior art or information and of its materiality; and (3) failure by the applicant to disclose the art or information resulting from an intent to mislead the PTO.[56]

An allegation of inequitable conduct can be rebutted by showing that (a) the prior art or information was not material, (b) if it was material, applicant did not know about it, (c) if the applicant knew about it, they did not know of its materiality, or (d) that the failure was not as a result of an intent to mislead the PTO.[57] For example, if the information is material, but was not disclosed, the failure to disclose will not alone support a finding of inequitable conduct if the reference is "simply cumulative to other references."[58]

In *In re Harita,* after a discussion with the inventor about whether or not to inform the U.S. attorney of a newly discovered material reference, a Japanese foreign associate who did not know of the duty of disclosure in U.S. practice advised the inventor not to forward the document.[59] Months after grant of the patent, the inventor discovered that the advice given by the Japanese associate was erroneous. Hence, the applicant submitted the document to the Patent Office and filed for reissue to narrow the scope of claims in order to avoid the prior art. A Special Program Examiner at the Patent Office rejected the request for reissue on the sole ground that the Japanese attorney did not disclose the newly discovered art to the Patent Office before the patent had issued.[60] The Examiner claimed that there was an intent to mislead which constituted 'inequitable conduct,' and thus based the rejection on 37 C.F.R. § 1.56 (d).

On appeal, the court noted that the events occurred from 1974–1976, before the adoption of Rule 56. The court held that the case must be considered in the light of the situation as it existed when the acts took place. The rule that was in existence at that time stated that any application filed fraudulently or in connection with fraud on the Patent Office may be stricken from the files. To constitute fraud, both materiality of the prior art and intent to act inequitably so as to mislead the Patent

56 *See* FMC Corp. v. Manitowoc Co., Inc., 835 F.2d 1411, 5 U.S.P.Q.2d 1112 (Fed. Cir. 1987); Laitram Corp. v. Cambridge Wire Cloth Co., 785 F.2d 292, 294 (Fed. Cir.). *cert. denied*, 479 U.S. 820, 107 S. Ct. 85 (1986); Braun Inc. v. Dynamics Corp. of America, 975 F.2d 815, 822 (Fed. Cir. 1992); Prevue Interactive, Inc. v. Starsight Telecast, Inc., 1999 U.S. Dist. LEXIS 1956.

57 *See* case cited *supra* note 58.

58 *See* Scripps Clinic & Research Found. v. Genentech, Inc., 927 F.2d 1565, 1582, 18 U.S.P.Q.2d 1001, 1014 (Fed. Cir. 1991).

59 *See* case cited *supra* note 51.

60 *See id.*

Office must be established. In this case, materiality was not contested thus the only issue was the element of intent. It was determined that the Japanese associate did not have the requisite intent to mislead, despite his gross negligence, due to the particular circumstances of the case including the fact that prior art did not have to be disclosed to the Japanese Patent Office after filing the application in Japan and the fact that the associate was very inexperienced with U.S. patent laws. The court ultimately held that the reissue should be granted.[61]

2. *Gross Negligence & Inference of Intent to Deceive*

Intent to act inequitably is a required element of inequitable conduct which is rarely presumed.[62] Although not directly concerned with prior art, the Court of Appeals in *Kingsdown Medical Consultants, Ltd. v. Hollister Inc.*, the Federal Circuit declined to infer a finding of deceitful intent where an attorney was grossly negligent during prosecution of an application by mis-listing the correspondence between the allowed claims of a parent and the amended claims of its continuation.[63] The holding was that gross negligence does not of itself justify an inference of intent to deceive.[64] Additionally, the court held that negligence can support an inference of intent only when, "viewed in light of all the evidence, including any evidence indicative of good faith," the negligent conduct is culpable enough "to require a finding of intent to deceive."[65] This was later confirmed by the Court of Appeals for the Federal Circuit in *Manville Sales Corp. v. Paramount Systems Inc.*[66]

One example which is related to submission of prior art had an interesting outcome. In *Halliburton Co. v. Schlumberger Technology Corp.*, the plaintiff sued Schlumberger Technology Corp. for patent infringement.[67] The district court found that, during the application process, the Examiner had cited six patents but that Halliburton had not disclosed any prior art to the Patent Office. In agreement with Schlumberger, the court determined that Halliburton should have cited the seven other prior art references of which they were aware. The district court determined that this failure to disclose constituted inequitable conduct which led to the enforceability the patents in question. On appeal, the

61 *See id.*
62 *See id.*
63 *See id.*
64 *See id.*
65 *Id.* at 807.
66 *See* Manville Sales Corp. v. Paramount Systems Inc., 917 F.2d 544, 16 U.S.P.Q.2d 1587 (Fed. Cir. 1990).
67 *See* case cited *supra* note 54.

Court of Appeals held that the analysis of such a case required a showing of threshold levels of materiality and intent and the application of the balancing test enunciated in *J.P. Stevens*.[68] This court reasoned that, because the references cited by the Examiner were more closely related to the patent claims than the uncited prior art, the latter were cumulative and did not need to be disclosed. The element of intent was then addressed as follows: although Halliburton was aware of the withheld references, their counsel did not consider the references material. The Court of Appeals held that counsel's assertion that he did not intend to mislead was objectively reasonable, and, therefore, that, despite his gross negligence, he did not engage in inequitable conduct.[69]

3. *Antitrust Implications*

Although the hurdles involved in proving inequitable conduct are substantial, once proven, they can lead to antitrust implications for patent owners. Patent owners who sue for infringement may incur antitrust liability for enforcement of a patent known to be obtained or maintained through fraud.[70] In *Walker Process Equipment, Inc. v. Food Machinery & Chemical Corp.*, Walker Process Equipment, Inc., the defendant in a patent infringement suit, filed a counterclaim which alleged that the plaintiff, Food Machinery & Chemical Corp., had illegally monopolized interstate and foreign commerce through a patent obtained and maintained fraudulently and in bad faith.[71] Walker Process claimed that such a monopoly would be in violation of § 2 of the Sherman Act (15 U.S.C. § 2), under which a successful party is entitled to treble damages under § 4 of the Clayton Act (15 U.S.C. § 15). Prior common law dictated that only the government, not private parties, possessed the statutory authority to bring suit for such a claim. Walker Process claimed that this action was justified based on the fact that the existence of the plaintiff's patent deprived the defendant of business it would have otherwise enjoyed. The Supreme Court agreed and held that this action could be brought by a private party in cases involving patents procured by intentional fraud, that is, by knowingly and willfully misrepresenting facts or willfully withholding information in dealings with the Patent Office.[72]

---

68 *See* case cited *supra* note 17.

69 *See* case cited *supra* note 54.

70 *See* Atari Games Corp. v. Nintendo of America, Inc., 897 F.2d 1572, 14 U.S.P.Q.2d 1034 (Fed. Cir. 1990).

71 *See* Walker Process Equipment, Inc. v. Food Machinery & Chemical Corp., 382 U.S. 172, 147 U.S.P.Q. (BNA) 404 (1965).

72 *See* cases cited *supra* note 74–75.

A *Walker Process* antitrust claim therefore requires proof of intentional fraud.[73] Although inequitable conduct may render a patent unenforceable, this differs from the type of fraud required to support a *Walker Process* type antitrust claim.[74] A finding of *Walker Process* fraud requires higher threshold showings of both materiality and intent than does a finding of inequitable conduct.[75]

VI. Conclusion

In summary, a patent is better able to withstand challenge in litigation and enjoys a higher standard of review when one or more Information Disclosure Statements have been filed during the prosecution of the application. Additionally, the statutory duty to disclose information material to the patentability of any claim in the application is owed by every individual associated with the prosecution of a patent application and is only discharged by filing proper and complete Information Disclosure Statements. All those substantively associated with the preparation and prosecution would be wise to protect their future patent from the beginning of the application's prosecution by timely filing a comprehensive IDS in the proper form and citing all material prior art.

73 *See* FMC Corp. v. Manitowoc Co., Inc., 835 F.2d 1411, 5 U.S.P.Q.2d 1112 (Fed. Cir. 1987).

74 *See* FMC Corp. v. Hennessy Industries Co., Inc., 836 F.2d 521, 5 U.S.P.Q.2d 1272 (Fed. Cir. 1987).

75 *See* Nobelpharma AB v. Implant Innovations, Inc., 141 F.3d 1059,46 U.S.P.Q.2d 1097 (Fed. Cir. 1998).

Radio Free PTO
PATENT OFFICE PROFESSIONAL ASSOCIATION

Important 690E Codes

- **Search and Examination**
  - Personal Interview (112024)
  - Appeal Conference (112041)
  - Travel Time (119070)
  - Reading/Classifying Technical Literature (112036)
  - Inspection and Study of Documents (011094)
  - CAFC Hearings (112036)
  - Updating MPEP (119020)
  - Markush Searching (112025)
  - Other Examining Activities (112036)
- **Automation/Computer**
  - Catostrophic Computer Failure (090180)
  - Network Problems Preventing Use of ActionWriter (112036)
  - Network Problems Preventing Use of Image Searching (011094)
  - Compensation for Automated Search Tools Transition (119500)
  - Developing Automated Search Tools Training (119501)
  - Conduting Automated Search Tools Training (119502)
- **Lectures, Seminars, and Trips**
  - Examiner Field Trip (119033)
  - Remote Technical Field Trip (090133)
  - Local Technical Field Trip (090134)
  - 16/40 Program (090130)
  - Legal Lecture Series(090132)
  - Other Seminars (090131)
- Training
  - In-House Legal Training (090110)
  - In-House Technical Training (090140)
  - Technical Courses Program (090141)
  - On the Job Training in a Foreign Technical Area (090140)
  - Specialty Training (090142)
  - Ethics Training (090117)
  - Commercial Database Training (114002)
  - P.E.I.T. Student (090120)
  - P.E.I.T. Instructor (090121)
  - Phase II,III,IV Student (090122)
  - Phase II,III,IV Instructor (090123)
- Union Activities
  - Attending POPA Meeting Once a Year (090285)
  - Attending POPA Meeting for Discussion of Mid-Term Changes (090302)
  - Preparation of Grievance by Grievant (090291)
- EEO Activities
  - Prosecution of Own EEO Complaint (090401)
  - Contacting EEO Counselor and Explaining Complaint (090403)
  - Serving as Complainant's Representative (090402)
  - Serving as Witness During EEO Investigation (090404)
  - EEO Committee Members Time Charged to EEO (090403)
  - Serving as Attorney for PTO at EEO Hearings (090405)
  - EEO Counseling (090406)
  - Pre-Complaint Processing by Supervisor (090408)
  - Formal Complaint Processing by Supervisor (090407)
- Miscellaneous
  - Office Move (090700)
  - Fire Drill (011094)
  - Disturbance Time (090209)
  - Donating Blood (051200)

- Completion of Financial Disclosure Statement (011095)
- U.S. Bond Drive (011094)
- Combined Federal Campaign (011094)
- Discussion Group Meeting (090274)
- PTOS Activities(090231)
- PTOS Annual Meeting (090241)
- Inspector General Interview (090152)

If You Lose Work On The Computer System

There is a past practice established in Groups 2200 and 2100 that allows you to take **CATASTROPHIC TIME** every time you lose work on the computer system. You must c Desk (305-9000) and get a control number. This number must be put in the remarks s 690E and you should claim the actual work time that had to be repeated because of th failure. The Subproject Number to put on the 690E is 090180. You can see (and print) the Don Kelly memo dated August 8, 1990 that established the past practice by clicki your SPE will not approve the time, see your nearest POPA representative.

IBM CORPORATION
INTELLECTUAL PROPERTY LAW DEPARTMENT - 4054
11400 BURNET ROAD
AUSTIN, TEXAS 78758
FAX # 512-838-3516

DATE: 1/4/00

Number of Pages to Follow (including cover sheet) 9

SEND TO:

Name: Examiner J. Homere USPTO, Group 2177
Tel No: 703-308-6647
FAX #: 703-308-6606

FROM:

Name: Volel Emile
Tel. No.: 512-823-1005 ✓
Contact #: 512-8

SPECIAL INSTRUCTIONS OR COMMENTS:

Proposed draft Final amendment
for 09/118,555

1/4/01

PATENT
09/118,555

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of:
J. R. Dean et al.
Serial No:09/118,555

Filed: 7/17/98

Title:Automatic Cleanup of User Data in a Network Environment

Before the Examiner:
J. Homere
Group Art Unit:2777
Intellectual Property
Law Department
International Business
Machines Corporation
11400 Burnet Road
Austin, Texas 78758

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Box Non-Fee Amendments (PATS), Assistant Commissioner of Patents Washington, D. C. 20231 on ______________.

Barbara Rogers ______________

______________ ______________
Signature Date

PROPOSED DRAFT AMENDMENT AFTER FINAL REJECTION
UNDER 37 CFR 1.116

Assistant Commissioner of Patents

Washington, D. C. 20231

Sir:

In response to the Final Office Action of December 19, 2000, please consider the following Remarks.

IN THE CLAIMS:

For the Examiner's convenience all pending claims for which reconsideration is requested are provided below.

1. (Amended) A method of identifying files belonging to a user in a multi-user system comprising the steps of:

   creating a data base file, said data base file including a list of all users and a list of all files in said system; and

   associating each file belonging to a user with said user.

2. (Amended) The method of Claim 1 wherein said list of all files includes storage location of each file.

3. (Amended) The method of Claim 2 further comprising the step of periodically updating said data base file.

4. (Amended) The method of Claim 19 wherein said step of periodically updating said data base file includes the step of ascertaining whether any one of the files listed in said data base file is deleted.

5. The method of Claim 4 wherein if a file is deleted, said data base file is updated by deleting said file from said list of files.

6. (Amended) The method of Claim 5 wherein if a user of said multi-user system no longer has access to said system, using said data base file to delete all files belonging to said user.

7. (Amended) A multi-user computer system having means for identifying files belonging to a user of said multi-user system comprising:

at least a storage space for storing files;

means for creating a data base file, said data base file including a list of all users and a list of all files in said system;

means for associating each file belonging to a user with said user; and

means for storing said data base file in said storage space.

8. (Amended) The multi-user computer system of Claim 7 wherein said list of all files includes storage location of each file.

9. (Amended) The multi-user computer system of Claim 8 further comprising means for periodically updating said data base file.

10. (Amended) The multi-user computer system of Claim 20 wherein the step for periodically updatingsaid data base file includes means for ascertaining whether any one of the files listed in said data base file is deleted.

11. The multi-user computer system of Claim 10 wherein if a file is deleted, said data base file is updated by deleting said file from said list of files.

12. (Amended) The multi-user computer system of Claim 11 wherein if a user of said multi-user system no longer has access to said system, using said data base file to delete all files belonging to said user.

13. (Amended) A computer program having embedded in a computer readable medium for identifying files belonging to a user in a multi-user system comprising:

program code means for creating a data base file, said data base file including a list of all users and a list of all files in said system; and

program code means for associating each file belonging to a user with said user.

14. (Amended) The computer program of Claim 13 wherein said list of files includes storage location of each [and every single] file.

15. (Amended) The computer program of Claim 14 further comprising program code means for periodically updating said data base file.

16. (Amended) The computer program of Claim 21 wherein said program code means for peridocally updating said data base file includes program code means for ascertaining whether any one of the files listed in said data base file is deleted.

17. The computer program of Claim 16 wherein if a file is deleted, said data base file is updated by deleting said file from said list of files.

18. (Amended) The computer program of Claim 17 wherein if a user of said multi-user system no longer has access to said system, using said data base file to delete all files belonging to said user.

19. The method of Claim 3 wherein the step of periodically updating said data base file includes updating said data base file each time a user stores or saves a file in said multi-user system.

20. The multi-user computer system of Claim 9 wherein the means for periodically updating said data base file includes means for updating said data base file each time a user stores or saves a file in said multi-user system.

21. The computer program of Claim 15 wherein the program code means for periodically updating said data base file includes program code means for updating said data base file each time a user stores or saves a file in said multi-user system.

REMARKS

Claims 1 - 21 are pending in the present Application. In the above-identified Office Action, the Examiner rejected Claims 1 - 21 under 35 U.S.C. §102(e) as being anticipated by Cannon et al.

For the reasons stated more fully below, Applicants submit that the claims of the Application are allowable over the applied reference. Hence, reconsideration, allowance and passage to issue are respectfully requested.

As stated in Applicants' previous Response, the present invention provides a method of identifying files belonging to a user in a multi-user system. According to the teachings of the invention, a database file containing a list of all the users in the multi-user system is created. The database file also includes a list of all the files in the system. All the files that belong to a user are associated with that user. This then allows for files, that should be deleted from the system as a result of a user not having anymore access to the system, to be easily identified as such.

The invention is set forth in claims of varying scopes of which Claim 1 is illustrative.

> 1. A method of identifying files belonging to a user in a multi-user system comprising the steps of:
> ***creating a data base file, said data base file including a list of all users and a list of all files in said system; and***
> ***associating each file belonging to a user with said user***. (Emphasis added.)

The Examiner rejected the claims under 35 U.S.C §102(e) as being anticipated by Cannon et al. Applicants respecfully disagree.

Cannon et al. disclose a storage management system with file aggregation and space reclamation within aggregated files. According to Cannon et al., the storage management system is set in a client-server system. Each client file (or user file as it is referred by Cannon et al.) is stored in the client-server system's database. An inventory table (i.e., Table 1) is used to cross-reference each user file with a client number and a client type. The client number identifies the originating client station and the client type identifies the type of computer is at the client station or the operating system being run on the client station. (See col. 7, lines 47 - 53.)

However, Cannon et al. do not teach or show each file being cross-referenced with a user as claimed. This distinguishing factor is very important. For example, suppose there was an employee (or $user_1$) assigned to client $station_1$. Suppose further that after $user_1$ has left the company, $user_2$ is assigned to client $station_1$. Then, according to the present invention, an admisnistrator would be able at any time to single out all the files that belong to $user_1$ and all those that belong to $user_2$. The disclosure of Cannon et al., on the other hand, would only allow an administrator to determine whether a file originated from client $station_1$, but, would not allow the true owner of the file to be known. That is, although the disclosure of Cannon et al. would allow a file to be identified as originated from client $station_1$, it would not allow one to determine whether the file belongs to $user_1$ or $user_2$.

The Examiner stated that Cannon et al. disclose a managed file (502) stored in relational database (13) having a list of users (a - p) and files (502a - 502p) corresponding thereto. Applicants disagree.

As shown in Table 1, Cannon et al. disclose a group of files "a - p" that reside in a relational database 502. Since there are a plurality of relational databases (i.e., databases 502, 504, 506 and 508), to distinguish between the different files residing in the different databases, Cannon et al. use the relational database in which the file resides in conjunction with the actual filename when referring to a file. For example, 502a represents file "a" residing in database 502. However, "a - p" do not represent a list of users as claimed by the Examiner.

Based on the foregoing, Applicants submit that Claim 1 and its dependent claims should be allowable. Claims 7 and 13 and their dependent claims, which all incorporate the above emboldened and italicized limitations, should be allowable as well. Consequently, reconsideration, allowance and passage to issue are once more respectfully requested.

Respectfully submitted,
J. R. Dean et al

By:________________________
Volel Emile
Attorney for Applicants
Registration No. 39,969
(512) 823-1005

From 90|005,727

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re patent of

Bruce Dickens	Group Art Unit: Unknown
Patent No. 5,806,063	Examiner: Unknown
Issued: September 8, 1998	

For: **DATE FORMATTING AND SORTING FOR DATES SPANNING THE TURN OF THE CENTURY**

REQUEST FOR REEXAMINATION

Sir:

Reexamination under 35 U.S.C. § 302 - 307 and 37 C.F.R. 1.510 is requested of United States patent number 5,806,063 which issued on September 8, 1998 to Bruce Dickens. This patent has not expired.

00905727.050600

I. Claims for which reexamination is requested:

Reexamination is requested of claims 1-15 of the Dickens patent in view of Clipper 5 A Developers Guide, by Booth et al. published in 1991, portions of which are attached hereto.

II. Explanation of pertinency and manner of applying cited prior art to every claim for which reexamination is requested based on prior art:

Claims 1-15 are fully anticipated under 35 U.S.C. § 102(b) by the prior Booth et al. publication. A detailed explanation of how each and every claim limitation is anticipated by the Booth et al. reference is provided below. The Booth et al. reference describes the Clipper programming language which is a general-purpose programming language with integrated libraries for screen management and database navigation.

Claim 1	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
1. A method of processing symbolic representations of dates stored in a database, comprising the steps of	Booth et al. describes the Clipper programing language which includes many functions for processing dates stored in a database. See page 939.
providing a database with symbolic representations of dates stored therein according to a format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator, and Y_1Y_2 is the numerical year designator, all of the symbolic representations of dates falling within a 10-decade period of time;	The Clipper programming language is used to manipulate databases. The functions described throughout the manual are described with the understanding that a database is present. For example, sorting a database is described on page 839. The database may include dates (date functions are described throughout the manual, for example see page 939). Storing dates in the symbolic representation mm/dd/yy is described on page 940. The two digit year representation requires all of the dates to be within a 100 year (10 decade) period. Furthermore, the SET EPOCH command, described on page 941 requires all of the dates to fall within a 10-decade window, since the dates are assumed to include only two digits for the year.

09005727.051600

selecting a 10-decade window with a Y_AY_B value for the first decade of the window, Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database;	The SET EPOCH command described beginning on page 941 is used to select a 10-decade window. The syntax for the command is "SET EPOCH to <nYear>." The 10-decade window is selected by assigning a value to <n Year>. Furthermore, the value of <nYear> determines the first year of the window and must therefore be no later than the earliest year in the database. A four digit value is input, which includes year digits corresponding to Y_AY_B.
determining a century designator C_1C_2 for each symbolic representation of a date in the database, C_1C_2 having a first value if Y_1Y_2 is less than Y_AY_B and having a second value if Y_1Y_2 is equal to or greater than Y_AY_B; and	The following description is given on page 941: "The SET EPOCH command informs the system how to handle dates that use only two digits for the year. When a two-digit year is entered into a date, its year digits are compared with the year digits of the epoch setting to determine the century to place the date into. If the two digits are prior to the setting of SET EPOCH, the year is assumed to be in the next century. If the digits are greater than or equal to the SET EPOCH setting, the year is assumed to be in the current century." Century designators are shown on page 942.

reformatting the symbolic representation of the date with the values C_1C_2,Y_1Y_2,M_1M_2, and D_1D_2 to facilitate further processing of the dates.[1]	The example given for the SET EPOCH command on page 942 shows that it reformats the date to add century designator values C_1C_2 of 19 and 20. In particular, a variable mdate is defined having a date value of "01/22/89." When SET EPOCH is set to 1900, the year is reformatted and displayed as 1989, which includes the century designator 19. When SET EPOCH is set to 1990, the year is reformatted and displayed as 2089, which includes the century designator 20. Once the date is properly interpreted to include a century designator with the SET EPOCH command, the dates can be further processed by any of the many processing commands. For example, at the top of page 939, Booth et al. describes performing math operations with dates.

Claim 2	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
2. The method of claim 1, wherein the 10-decade window includes the decade beginning in the year 2000.	Booth et al. teaches that the SET EPOCH command allows the user to specify the centuries and specific window to use. At the top of page 942, a window from 1990 to 2089 is disclosed, which includes "the decade beginning in the year 2000" (the years 2000-2010).

[1] See section III for a discussion of this limitation.

Claim 3	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
3. The method of claim 2, wherein the step of determining includes the step of determining the first value as 20 and the second value as 19.	The example given by Booth et al. at the top of page 942 uses a first century designator value of 20 and a second century designator value of 19.

Claim 4	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
4. The method of claim 1, including an additional step, after the step of reformatting, of sorting the symbolic representations of dates.	Booth et al. teaches sorting dates in many places. For example sorting databases is described on page 839. The user determines which field the database should be sorted on. Sorting dates is explicitly described in the middle of page 945. Dates may only be sorted after they have been interpreted in accordance with the SET EPOCH command described above.

Claim 5	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
5. The method of claim 1, wherein the step of reformatting includes the step of reformatting each symbolic representation of a date into the format $C_1C_2\ Y_1Y_2\ M_1M_2D_1D_2$.	On page 945, Booth et al. teaches representing dates in the format $C_1C_2\ Y_1Y_2\ M_1M_2D_1D_2$, for sorting and indexing. Mapping (reformatting) dates into the format $C_1C_2\ Y_1Y_2\ M_1M_2D_1D_2$ is also taught on page 941.

Claim 6	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
6. The method of claim 5, including an additional step, after the step of reformatting, of sorting the symbolic representations of dates using a numerical-order sort.	As stated above, sorting databases is described on page 839. Sorting in either ascending or descending numerical order is taught on page 840. Dates may only be sorted after they have been interpreted in accordance with the SET EPOCH command described above.

Claim 7	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
7. The method of claim 1, wherein the step of providing a database includes the step of converting pre-existing date information having a different format into the format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator and Y_1Y_2 is the numerical year designator.	Booth et al. describes the SET DATE command on page 939. The command allows the user to specify the format of dates. All of the formats shown on page 940 use a two digit representation for the month. Accordingly, it is apparent from the description of the SET DATE command that the user of the program will have to convert any date information which does not represent the month with two digits into such a representation. For example, if the user wishes to enter the date February 3, 1980, the user would enter a value of 02/03/80. The conversion step described above is an inherent feature of Booth et al. Moreover, it was conventional and notoriously well known to convert pre-existing dates into the format MMDDYY to facilitate computer processing, such as numerical sorting. Even Dickens admits that the format MM/DD/YY was a conventional format in col. 1, lines 33-35.

Claim 8	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
8. The method of claim 1, wherein the step of selecting includes the step of selecting Y_AY_B such that Y_B is 0 (zero).	Booth et al. teaches selecting a value corresponding to Y_AY_B equal to 90 in the example given at the top of page 942

Claim 9	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
9. The method of claim 1, including an additional step, after the step of reformatting, of storing the symbolic representation of dates and their associated information back into the database.	On page 841, Booth et al. teaches storing data which has been manipulated back into the database. An original database "CUSTOMER.DBF" is sorted and the sorted database is labeled "NEW_CUST.DBF." "CUSTOMER.DBF" is then renamed "CUSTOMER.OLD" and the sorted "NEW_CUST.DBF" database is renamed to the original "CUSTOMER.DBF." The manipulation of data may include reformatted into the format YYYYMMDD and sorting (see pages 945 and 941).

Claim 10	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
10. The method of claim 9, including the additional step, after the step of reformatting, of manipulating information in the database having the reformatted date information therein.	Booth et al. teaches manipulating data in a database throughout the manual. For example. on page 939, Booth et al. teaches that "Clipper includes very powerful date manipulation capabilities" and describes performing math operation on dates. Booth et al. discusses manipulating a database after storing data which has been manipulated back into the database on page 841. For example, Booth et al. state that " A sorted database should be used when you expect very little update activity to occur in the file." Before any manipulation would take place, the SET EPOCH command is used to interpret and reformat the date values.

Claim 11	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
11. A method of processing dates in a database, comprising the steps of	Booth et al. describes the Clipper programing language which includes many functions for processing dates stored in a database. See page 939.
providing a database with dates stored therein according to a format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator, and Y_1Y_2 is the numerical year designator, all of dates falling within a 10-decade period of time which includes the decade beginning in the year 2000;	The Clipper programming language is used to manipulate databases. The functions described throughout the manual are described with the understanding that a database is present. For example, sorting a database is described on page 839. The database may include dates (date functions are described throughout the manual). Storing dates in the format mm/dd/yy is described on page 940. The two digit year representation requires all of the dates to be within a 100 year (10 decade) period. Furthermore, the SET EPOCH command, described on page 941 requires all of the dates to fall within a 10-decade window, since the dates are assumed to include only two digits for the year. The SET EPOCH command allows the user to specify the centuries and specific window to use. At the top of page 942, a window from 1990 to 2089 is disclosed, which includes the decade beginning in the year 2000.

selecting a 10-decade window with a Y_AY_B value for the first decade of the window, Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database;	The SET EPOCH command described beginning on page 941 is used to select a 10-decade window. The syntax for the command is "SET EPOCH to <nYear>." The 10-decade window is selected by assigning a value to <n Year>. The value of <nYear> determines the first year of the window and must therefore be no later than the earliest year in the database. A four digit value is input, which includes year digits corresponding to Y_AY_B.
determining a century designator C_1C_2 for each date in the database, C_1C_2 having a first value if Y_1Y_2 is less than Y_AY_B and having a second value if Y_1Y_2 is equal to or greater than Y_AY_B;	The following description is given on page 941: "The SET EPOCH command informs the system how to handle dates that use only two digits for the year. When a two-digit year is entered into a date, its year digits are compared with the year digits of the epoch setting to determine the century to place the date into. If the two digits are prior to the setting of SET EPOCH, the year is assumed to be in the next century. If the digits are greater than or equal to the SET EPOCH setting, the year is assumed to be in the current century." Century designators are shown on page 942.

reformatting each date in the form $C_1C_2,Y_1Y_2,M_1M_2D_1D_2$ to facilitate further processing of the dates; and	The example given for the SET EPOCH command on page 942 shows reformatting each date by adding century designator values C_1C_2 of 19 and 20. On page 945, Booth et al. teaches representing dates in the format C_1C_2 Y_1Y_2 $M_1M_2D_1D_2$, for sorting and indexing. Mapping (reformatting) dates into the format C_1C_2 Y_1Y_2 $M_1M_2D_1D_2$ is also taught on page 941. Once the date is properly interpreted by including a century designator with the SET EPOCH command, the dates can be further processed by any of the many processing commands. For example, at the top of page 939, Booth et al. describe performing math operations with dates.
sorting the dates in the form $C_1C_2,Y_1Y_2,M_1M_2D_1D_2$.	Booth et al. teaches sorting in many places. For example sorting databases is described on page 839. The user determines which field the database should sorted on. Sorting dates in the form C_1C_2 Y_1Y_2 $M_1M_2D_1D_2$ is explicitly described in the middle of page 945.

Claim 12	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
12. The method of claim 11, wherein the step of providing a database includes the step of converting pre-existing date information having a different format into the format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator and Y_1Y_2 is the numerical year designator	Booth et al. describes the SET DATE command on page 939. The command allows the user to specify the format of dates. All of the formats shown on page 940 use a two digit representation for the month. Accordingly, it is apparent from the description of the SET DATE command that the user of the program will have to convert any date information which does not represent the month with two digits into such a representation. For example, if the user wishes to enter the date February 3, 1980, the user would enter a value of 02/03/80. The conversion step described above is an inherent feature of Booth et al. Moreover, it was conventional and notoriously well known to convert pre-existing dates into the format MMDDYY to facilitate computer processing, such as numerical sorting. Even Dickens admits that the format MM/DD/YY was a conventional format in col. 1, lines 33-35.

Claim 13	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
13. The method of claim 11, wherein the step of selecting includes the step of selecting Y_AY_B such that Y_B is 0 (zero).	Booth et al. teaches selecting a value corresponding to Y_AY_B equal to 90 in the example given at the top of page 942.

Claim 14	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
14. The method of claim 11, including an additional step, after the step of sorting, of storing the sorted dates and their associated information back into the database.	On page 841, Booth et al. teaches storing data which has been manipulated back into the database. An original database "CUSTOMER.DBF" is sorted and the sorted database is labeled "NEW_CUST.DBF." "CUSTOMER.DBF" is then renamed "CUSTOMER.OLD" and the sorted "NEW_CUST.DBF" database is renamed to the original "CUSTOMER.DBF." The manipulation of data may include reformatted into the format YYYYMMDD and sorting (see pages 945 and 941).

Claim 15	Implementation in Clipper 5 A Developer's Guide, by Booth et al. Published in 1991
15. The method of claim 14, including the additional step, after the step of sorting, of manipulating information in the database having the reformatted date therein.	Booth et al. teaches manipulating data in a database throughout the manual. For example. on page 939, Booth et al. teaches that "Clipper includes very powerful date manipulation capabilities" and describes performing math operations on dates. Booth et al. discusses manipulating a database after storing data which has been manipulated back into the database on page 841. For example, Booth et al. states that " A sorted database should be used when you expect very little update activity to occur in the file." Before any manipulation would take place, the SET EPOCH command is used to interpret and reformat the date values.

<u>III. Reformatting limitations</u>:

Both independent claims, claims 1 and 11, contain "reformatting" limitations. In the Office Action mailed November 17, 1997, the Examiner rejected all of the pending claims under 35 U.S.C. § 112, first paragraph, as based on a disclosure which is not enabling. The Examiner indicated that the conversion of existing symbolic date representations without the addition of new data fields was critical or essential to the practice of the invention and not enabled by the disclosure. The Examiner then indicated that the specification clearly involves additional digits to solve the Y2K problem.

After reviewing the windowing technique disclosed in <u>The Year 2000 and 2-Digit Dates: A Guide for Planning and Implementation</u>, Third Edition, May, 1996, by IBM Corp. provided to the Applicant by the Patent Office, the Applicant filed a response on March 17, 1998. In the first four pages of the response, the applicant summarized his invention. Beginning at the bottom of page 3, the Applicant stated the following:

> "However, the method of the present invention need not store the converted date in data storage. Instead, the original dates and data storage remain undisturbed. This aspect of the present invention thus allows conversion of dates to compensate for century designations without requiring the addition of data fields to permanently store the century designations."

On page 4 of the response, the Applicant responded to the rejection under 35 U.S.C. § 112, first paragraph, by stating that the method of the claimed invention does not require that the converted data that includes the century designations be stored in data storage. Furthermore, the Applicant filed a Supplemental Response on April 2, 1998. In this Response, the Applicant modified the claimed invention by removing language from most of the claims suggesting that the reformatted data was stored in the database. In the middle of page 4, the applicant made the following statement: "Claims 1 and 11 have been amended so as to <u>not</u> require storage of the converted dates, thereby not imposing any requirement for new data fields." <u>This statement is clear evidence that the original claimed invention required the storage of the converted dates</u>.

The Applicant and the Examiner conducted a telephone interview on April 2, 1998. The parties agreed that "The Summary of the Invention, and the arguments of the Response, were not entirely in conformity with the claims, which would be potentially allowable if the use of the additional century digits do not include their storage in the data base." The Interview Summary also indicates that it was further agreed that the Applicant would fax a supplementary amendment to address this problem. A Notice of Allowance was mailed on April 8, 1998. As part of the Notice of Allowability, the Examiner objected to the drawings. The Examiner indicated that "FIG. 2 does not conform to the claims <u>as amended</u>. In particular, it shows box 36 labeled: reformat data in data base" (emphasis added). The Examiner went on to state that the Summary of the Invention, the statements at the bottom of page 3 of the response, and the reasons for allowance below specifically preclude this step and that the box should be removed from the drawings. The applicant submitted formal drawings which removed block 36 from FIG. 2. Block 36 included the label "Reformat dates in data base."

The Requester respectfully submits that the above excerpts from the prosecution history show that the Applicant attempted to modify his invention during the prosecution of the application. In particular, after reviewing the windowing technique disclosed in <u>The Year 2000 and 2-Digit Dates: A Guide for Planning and Implementation</u>, Third Edition, May, 1996, by IBM Corp., the Applicant realized how impractical his invention was and attempted to suggest that the invention did not require the modification of data in a database. To the extent that amending the original claimed invention from including limitations such as "reformatting the symbolic representation of the date <u>in the database</u>" to the new claimed invention including new limitations such as "reformatting the symbolic representation of the date," is not prohibited new matter, the Requester respectfully submits that the limitation of "reformatting the symbolic representation of the date" is properly interpreted to require storing the converted data in the database. The Booth et al. reference anticipates the original invention, as originally claimed, and the new invention, as presently claimed.

<u>IV. Statement pointing out substantial new question of patentability</u>:

The Booth et al. reference referred to above was not of record in the file of the Dickens patent. Since claims 1-15 in the Dickens patent are not patentable over this document, a substantial new question of patentability is raised. Furthermore, the Booth et al. reference is closer to the subject matter claimed by Dickens than any prior art which was cited during the prosecution of the Dickens patent. The Booth et al. reference provides teachings that were not provided during prosecution of the Dickens patent.

[illegible]

Ross F. Hunt, Jr.
Registration No. 24,082

Comm'r

DECISION

Pursuant to 37 CFR § 1.520, the Commissioner of Patents and Trademarks has determined that the prior art discussed below raises a substantial new question of patentability as to claims 1-15 of U.S. Patent No. 5,806,063 .

Claims 1-15 of U.S. patent No. 5,806,063, as a whole, are drawn to a method particularly useful for extending processing dates stored in a database beyond the turn of the century. The method comprises the steps of providing a database with dates stored in it according to a format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator, and Y_1Y_2 is the numerical year designator, all of the dates falling within a 10 decade period of time. A 10 decade window with a Y_AY_B value for the first year of the 10 decade window is selected, Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database. A century designator C_1C_2 is determined for each date in the database, C_1C_2 having a first value if Y_1Y_2 is less than Y_AY_B and having a second value if Y_1Y_2 is equal to or greater than Y_AY_B. Each date in the database is assigned the values C_1C_2 ,Y_1Y_2 ,M_1M_2, and D_1D_2 . In one case that produces particularly advantageous results for many operations, such as chronological date sorting, the date can then be represented in the form $C_1C_2Y_1Y_2M_1M_2D_1D_2$.

The '063 patent points out that, for the case of most practical interest, the 10 decade period of time (window) spans the year 2000 (i.e., includes decades earlier than the year 2000 and decades later than the year 2000), and it begins with a year in which the second digit of Y_AY_B is 0 (zero). Further, assume that a decade beginning in 1950 is selected as the first decade of the 10-decade window, i.e., Y_A is selected as 5. Then, C_1C_2 is assigned "20" if Y_1 is less than 5 and is assigned "19" if Y_1 is equal to or greater than 5. In that case, if Y_1Y_2 is "43", the century designator C_1C_2 is "20," indicating that the year in question in the database is 2043. On the other hand, if Y_1Y_2 is "63", the century designator C_1C_2 is "19," indicating that the year in question in the database is 1963. For chronological date sorting, the date is represented in the form $C_1C_2Y_1Y_2M_1M_2D_1D_2$; thus, the date 12/15/93 (Dec. 15, 1993) is represented as 19931215 and the date 12/15/00 (Dec. 15, 2000) as 20001215.

The Ohms Article:

The article B. G. Ohms, Computer Processing of Dates Outside the Twentieth Century, IBM Systems Journal, Vol. 25, No. 2, 1986 is relevant to claims 1-15 of the '063 patent. As background to explaining the relevance of the Ohms article, the broadest reasonable interpretation of the claims is set forth as follows:

There are eight parameters listed in claim 1, in the form of four pairs. In the absence of any separation of the elements of the pairs, as for instance Y_1 from Y_2 in Y_1Y_2, the pair is taken to designate a two-digit single value. Thus $Y_1Y_2 = 43$, not $Y_1 = 4$ and $Y_2 = 3$, and so, this notation is equivalent to a year designation as YY, which is inherently ordered as Y_1Y_2 by the positional nature of the decimal system.

In the absence of the use of individual decades, a 10-decade window is taken to be equivalent to a century window. Similarly, the use of Y_B as zero in some of the claims is taken as an arbitrary choice because there is no claim of use of this particular value, as there would be if the fast-sort techniques noted in the specification were claimed.

In light of column 3, lines 35-38, the selection of Y_AY_B in the claim is taken to be external to the part of the method which is automated. The determination of the earliest year date is taken to include the possibility that the user may have prior information about this number when Y_AY_B is chosen, and thus it is not required to include a search for this number within the method.

The reformatting of claim 1 is not limited to the specific one recited in claim 5.

Furthermore, the '063 patent claims can be reasonably interpreted such that the date formats that include the century digits C_1C_2 need not be stored into the database; the reformatting can be dynamic and created for the purpose of processing dates extracted from the database. Thus, in claim 1, for instance, the century designator is stated (in the claim) to be "for each symbolic representation of the date in the database;" however, it is not considered to be part of the symbolic representation but is related to it. The storage of "associated information" as in claim 9 is not required to be the century designation C_1C_2, and is not to occupy the date field(s).

Thus, the Ohms article is relevant as follows:

<u>**As to independent claim 1 of the '063 patent:**</u>

> **"A method of processing symbolic representations of dates stored in a database, comprising the steps of:"**

See the Ohms abstract at page 244, left hand column.

> **"Providing a database with symbolic representation of dates stored therein according to a format wherein M_1M_2 is the numerical month designator, D_1D_2 is the numerical day designator, and Y_1Y_2 is the numerical year designator, all of the symbolic representations of dates falling within a 10-decade period of time;"**

The eight parameters of the claim are not explicit in Ohms, which uses simply YYMMDD for the short Gregorian format, table 1 at page 247. As noted above, the YYMMDD of Ohms is equivalent to the $Y_1Y_2M_1M_2D_1D_2$ of the claim because the claim does not require a separate use of the digits.

> **"Selecting a 10-decade window with a Y_AY_B value for the first decade of the window, Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database;**

See Ohms page 248, right hand column, lines 10-18, where a year is specified as "the starting point of a 100-year range." In many cases, the range of dates in a database is known, and dates of only one century occur in the database. In others, the effective four year range of (2-digit) dates in the 20th century is known. In both cases, Y_AY_B can be selected to separate the dates of one century from another by choosing it to be less than or equal to Y_1Y_2.

The skilled artisan at the time of the invention would have been motivated to select the pivot Y_AY_B less than or equal to Y_1Y_2 because it would separate 2-digit dates into the proper century in an efficient manner by simply comparing them to Y_AY_B.

> **"Determining a century designator C_1C_2 for each symbolic representation of a date in the database, C_1C_2 having a first value if Y_1Y_2 is less than Y_AY_B and having a second value if Y_1Y_2 is equal to or greater than Y_AY_B."**

See Ohms page 248, right hand column, lines 10-18, where the effect of the pivot 25 determines 1925 while it also determines 2024, (from the dates with year digits of 25 and 24).

The distinction between the earliest Y_1Y_2 and Y_AY_B allows for the use of convenient separators, such as the 80-year window noted at page 248, right hand column, lines 28-38. The skilled artisan at the time of the invention would have been motivated to provide that Y_AY_B not be equal to the earliest Y_1Y_2 because it permits a convenient way of segregating the data between two centuries.

<u>As to claims 2-3 of the '063 patent:</u>

The examples of Ohms include the decade beginning in the year 2000, and the C_1C_2 values of 19 and 20.

<u>As to claim 4 of the '063 patent:</u>

Although the sorting of the symbolic values is not specifically addressed in Ohms; it would have been readily apparent to sort the date values, because this makes it possible to efficiently prioritize records that include dates, such as orders to be filled.

<u>As to claims 5-6 of the '063 patent:</u>

Ohms converts to a Julian date with leading 4-digit year from a 2-digit year format. This places the century first and makes sorting more efficient, and in particular places the 20th century dates before the 21st century dates, as they should be. This conversion would be equally applied to the Gregorian dates for the same reason Ohms applies it to the Julian dates.

<u>As to claim 7 of the '063 patent:</u>

The skilled artisan would have been motivated to preconvert databases not in the format required for the method of claim 1 in order to avoid the need for development of a separate system.

As to claim 8 of the '063 patent:

Ohms does not specify a value of $Y_B = 0$; however, the use of a particular second digit Y_B has no criticality in the claimed invention, - it is merely one of the 10 possible choices. Absent a showing of criticality, this would have been well within knowledge and level of skill of the ordinary artisan.

As to claim 9 of the '063 patent:

As to the claim 9 storage, see Ohms, page 248, right hand column, near the bottom, where it states with respect to storage "Of course, in the vast majority of cases, that is exactly what does take place."

As to claim 10 of the '063 patent:

The skilled artisan would have been motivated to manipulate the information in the database, because the point of the conversion is to provide for manipulating the information to avoid a Y2K problem.

Claims 11-15:

Claim 11 corresponds to claims 4-5 combined, claim 12 to claims 1+ 4+7, claim 13 to claims 1+4+8, claim 14 to 1+4+9, claim 15 to 1+4+10. Thus, Ohms is relevant to the elements of claims 11-15 accordingly, based on the analysis set forth above.

The Shaughnessy patent:

Shaughnessy patent 5,630,118 is generally directed to software patches of application programs, and in particular to providing a basis for proper subtraction of dates. [Abstract] The patent determines date pivots within a century span by using the current date [FIG 2,4], and install date [FIG 4], and sometimes an offset from these [FIG 4, numeral **18**]. It describes a variety of date formats including that of CCYYMMDD, which is related to the more parameterized $C_1C_2Y_1Y_2M_1M_2D_1D_2$ date format of the '063 patent. The preferred conversion is to a YYYYMMDD format. There is a fairly broad discussion of date windowing within these constraints.

Shaughnessy addresses the use of the earliest date in a **database**, through an implicit assumption that this is related to the install date of an application program, and the (unspoken) possibility that such a program might be a data base management system (DBMS). It does allow for a fixed arbitrary date to be entered [col 6 lines 30-35], but the context of this is a database in which there are dates both above and below the chosen pivot date.

Conclusion:

In view of the above teachings, a substantial new question of patentability is raised as to claims 1-15 of U.S. Patent No. 5,806,063. Reexamination of U.S. Patent No. 5,806,063 is hereby ordered under 37 CFR § 1.520. All the patent claims will be reexamined.

[illegible]

Q. Todd Dickinson
Assistant Secretary of Commerce and
Commissioner of Patents and Trademarks

kenccomr\5806063.cmr

60005562.122466

TC 2700
(2777)

LEVIN & HAWES LLP
ATTORNEYS AT LAW INTELLECTUAL PROPERTY LAW

384 FOREST AVENUE SUITE 13
LAGUNA BEACH, CALIFORNIA 92651
TEL 949.497.7676
FAX 949.497.7679
www.lagunalaw.com

Patent ◆ Trademark ◆ Copyright
Trade Dress ◆ and Related Litigation

OIPE
FEB 1 8 2000
PATENT & TRADEMARK OFFICE

February 18, 2000

Via Express Mail EE874964202US

RECEIVED
MAR 0 1 2000
GROUP 2700

#5

Box: Reexam

Assistant Commissioner for Patents
Washington, DC 20231

Dear Commissioner:

Enclosed is the Response to Commissioner Order for Reexamination
Reexamination No. 90/005,592
United States Patent No. 5,806,063
Issued to Dickens on September 8, 1998
Attorney Docket No. 2039-500

This submission comprises:
- Response
- Power of Attorney
- 37 CFR §3.73 Certificate
- Proof of Mailing by Express Mail
- Self addressed postcard

If you have any questions, please do not hesitate to contact me.

Sincerely,

[illegible]

William C. Cray
Reg. No.: 27627

WCC/ns
Enclosures
cc: JEH
WEL

90005592.021800

February 18 , 2000

BOX: Reexam
Assistant Commissioner for Patents
Washington, DC 20231

OIPE
FEB 18 2000
PATENT & TRADEMARK OFFICE

RECEIVED
MAR 01 2000
GROUP 2700

CERTIFICATE OF MAILING UNDER 37 CFR § 1.10

Re: Reexamination 90/005,592
United States Patent No. 5,806,063
Issued to Dickens on September 8,1998
Group art Unit: 2771
Examiner: Wayne Amsbury

Attorney Docket No. 2039-300

90005592 .021800

I hereby certify that the following attached items of correspondence comprising:

- Response
- Power of Attorney
- 37 CFR §3.73 Declaration
- Self addressed postcard

are being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR § 1.10 on the date indicated above and is addressed to:

BOX:

Assistant Commissioner of Patents
Washington, D.C. 20231

Express Mail Label Number: EE 874964202US
Date of Deposit: 2-18-2000

Nanees Salama
Nanees Salama

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Reexamination No. 90/005,592	
Ordered December 21, 1999	**Art Unit:** 2771
Patent No. 5,806,063	**Examiner:** Wayne Amsbury
Patentee: Dickens	**Attorney Docket No.:** 2039-500
Patent Owner: Dickens-Soeder2000, LLC	

Patent Owner's Response to the Commissioner Initiated Order for Reexamination

Honorable Commissioner of Patents
Washington, D.C. 20231

Dear Sir:

In response to the Commissioner Initiated Order for Reexamination dated December 21, 1999 ("the Order") in the above referenced Reexamination Proceeding, the patent owner submits that this Reexamination was not properly ordered and should be dismissed, or at least stayed in favor of going forward with the patent owner's Reissue Application to reissue the above referenced patent, filed contemporaneously with this Response.

The Order does not raise any substantial new question of patentability as to the above referenced patent, issued to Dickens on September 8, 1998 on an application filed on October 3, 1996, entitled DATE FORMATTING AND SORTING FOR DATES SPANNING THE TURN OF THE CENTURY ("the patent"). The Commissioner has cited B. Ohms, *Computer Processing of Dates Outside of the Twentieth Century*, IBM Systems Journal, Vol. 25, No. 2, 1986 ("Ohms"), which was not before the Examiner during the original prosecution of the patent, and United States Patent No. 5,630,118, issued to Shaughnessy on May 13, 1997, on an application filed on November 21, 1994, entitled, SYSTEM AND METHOD FOR MODIFYING AND OPERATING A COMPUTER SYSTEM TO PERFORM DATE OPERATIONS ON DATE FIELDS SPANNING CENTURIES ("the '118 patent), which was before the Examiner.

The Examiner was right to have issued the patent over the '118 patent, and Ohms adds nothing to change that original decision. The claims as originally issued in the patent are patentable over these references, alone or in combination. Any additional references can be considered within the Reissue proceeding.

As understood by the patent owner, the Commissioner's position is that Ohms renders the claims of the patent invalid for obviousness under 35 U.S.C. §103. The Commissioner is right not to rely on Ohms under 35 U.S.C. §102. Ohms also does not render the methods of the <u>claimed invention</u> obvious and in fact teaches away from the <u>claimed invention</u>.

The obviousness test under §103 must be applied to the subject matter of the <u>claimed invention as a whole</u>. There is no "gist" or "heart" or "core" of the invention evaluated for obviousness purposes. It is necessary to consider all of the subject matter of the <u>claimed invention</u>. Hawes, §7.04 at 7-6, *citing*, *Loctite Corp. v. Ultraseal Ltd.*, 781 F.2d 861, 228 U.S.P.Q. 90 (Fed. Cir. 1985). Even if the prior art references taken together show all of the elements of the <u>claimed invention</u>, it must still be shown that the <u>claimed invention</u> would have been obvious as a whole and it is improper to analyze the claimed invention by its separate parts, even if each is shown in the art. Hawes, §7.05, at 7-7, *citing, Custom Accessories, Inc. v. Jeffrey-Allan Industries, Inc.*, 807 F. 2d 955, 1 U.S.P.Q.2d 1197 (Fed. Cir. 1986), and *In re Wright*, 848 F.2d 1216, 6 U.S.P.Q.2d 1959 (Fed. Cir. 1988). In the present case, neither Ohms nor '118, nor the combination thereof, discloses all of the elements of the claimed invention, and, therefore, there does not even exist support a *prima facie* case for obviousness. *See*, MPEP § 2143.03, *In re Royka*, 490 F.2d 981, 180 U.S.P.Q. 580 (C.C.P.A 1974), and *In re Wilson*, 424 F.2d 1382, 1385, 165 F.2d 494, 496 (C.C.P.A. 1970). This alone is a basis for finding the claimed invention patentable over these references.

Hindsight must be avoided in combining references in the prior art. It is error to reconstruct the claimed invention from the prior art using the claimed invention as a blueprint. Hawes, §7.05, at 7-8, *citing, Panduit Corp. v. Dennison Manufacturing Co.*, 774 F.2d 1082, 227 U.S.P.Q. 337, 343 (Fed. Cir. 1985), *In re Find*, 837 F.2d 1071, 5 U.S.P.Q.2d 1596 (Fed. Cir. 1988), and *W.L. Gore & Associates, Inc. v. Garlock, Inc.*, 721 F.2d 1540, 220 U.S.P.Q. 303 (Fed. Cir. 1983), *cert. denied*, 469 U.S. 851 (1984). It is improper to reject a claim based upon the mere assertion that one of ordinary skill in the art would know to add a claimed feature to the claimed invention without the citation of a reference that teaches the claimed feature (MPEP §706.02(a)), or at least an affidavit from the Examiner detailing the Examiner's knowledge of the prior art under 37 C.F.R. §1.107(b). Hawes, §7.08, at 7-19, *citing, In re Newell*, 13 U.S.P.Q.2d 1248 (Fed. Cir. 1989), and *In re Kaplan*, 229 U.S.P.Q. 678, 683 (Fed. Cir. 1986).

A reference must be considered as a whole. A prior art reference that describes a product or process similar to the claimed product or process and also a statement that the product or process does not work does not teach the claimed product or process. A prior art reference may be considered to teach away when " a person of ordinary skill, upon reading the reference, would be discouraged from following the path set out in the reference, or would be led in a direction divergent from the path that was taken by the [inventor]." Hawes, §7.05, at 7-7, *citing*, *In re Gurley*, 27 F.3d 551, 553, 31 U.S.P.Q.2d 1130, 1131 (Fed. Cir. 1994).

<u>Ohms</u>

Ohms teaches a "[m]ethod[] of *using existing date formats across century boundaries .... The use of a format termed the Lilian date format ... is introduced.*" (Ohms, at 244, Abstract) Ohms teaches that "[t]he two positions traditionally used in both Julian and Gregorian date formats implicitly represent a year within a century. However, this system is inadequate for representing dates in more than one century." (*Id.* at 245) As a solution Ohms proposes a "Lilian date format [to] avoid[] the ambiguity by *using seven positions* for the *number of the days from the beginning of the Gregorian calendar, October 15, 1582.*" (*Id.* at 245) "The value is incremented by one for each subsequent day." (*Id.* at 246) Ohms explains that "the *Lilian date format* is presented here as the *basis for making date conversions ....* This format *handles processing across century years and other aspects of date conversion not currently adaptable to computer programming.*" (*Id.* at 244-45)

In this context, of database conversion to Lilian format from more traditional Gregorian or Julian formats, Ohms describes under the heading "Accommodating end users" the fact that they "*usually <u>enter</u> two digits for the year in a date and understand the ambiguity that this represents.*" (*Id.* at 248) Ohms goes on to say that:

to avoid adverse user reaction, [by requiring the entry of date data in other than two digits] programs must continue to function with only two digits for year. The inference of the year 1997 from 97 and 2003 from 03 must continue. For the exceptional case where the correct meaning could be 1897 and 1903, entry of all four digits may be required. (*Id.* at 248)

It is in this context that Ohm notes:

> it may be necessary to provide a *conversion function* that receives a definition of the implied century as a parameter. An excellent way to do this unambiguously is to specify a year as the desired starting point of a 100-year range. For example, if the starting year for the range is specified as 1925, dates with year digits between 25 and 99 would be between 1925 and 1999, and dates with year digits of 00 through 24 would lie between 2000 and 2024. (*Id.* at 248)

Ohms therefore, simply teaches storing dates in a database in Lilian format which "handles processing across century years" and "[a]ccommodating end users" who "enter two digits for the year" by "providing a conversion function" using windowing for data entry. Simply because the claimed invention employs a known technique, i.e., windowing, does not, of itself, make the process of the claimed invention obvious. Hawes §7.05, at 7-8.1, *citing*, *In re Brower*, 77 F.3d 422, 37 U.S.P.Q.2d 1663 (Fed. Cir. 1994).

There is, therefore, no teaching or suggestion in Ohms of:

providing a database with symbolic representations of dates stored therein according to a format wherein $M_1 M_2$ is the numerical month designator, $D_1 D_2$ is the numerical day designator, and Y_1 Y_2 is the numerical year designator, all of the symbolic representations of dates falling within a 10-decade period of time; … .

Unlike this recitation of claim 1, Ohms teaches providing a database with the dates in a Lilian format.

There is also, therefore, no teaching or suggestion in Ohms of:

providing a database … *all of the symbolic representations of dates falling within a 10-decade period of time*; … .

Ohms teaches having data in the database in Lilian format, i.e., within a ninety-nine million day window (seven chronological day date numbers starting at a given date).

There is also, therefore, no teaching or suggestion in Ohms of:

selecting a 10-decade window with a Y_A Y_B value for the first decade of the window, Y_A Y_B *being no later than the earliest Y_1 Y_2 year designator in the database*; … .

At best Ohms teaches or suggests selecting a Y_AY_B based upon dates that are currently being input into the database.

There is also, therefore, no teaching or suggestion in Ohms of:

determining a century designator C_1 C_2 for each symbolic representation of a date in the database, C_1 C_2 having … ; … .

Ohms teaches entering date data into the database to be converted into Lilian format for storage and manipulation within the database. He does not teach or suggest determining a century designator for data in the database. Lilian format needs none.

There is also, therefore, no teaching in Ohms of:

09005592.021800

reformatting the symbolic representation of the date with the values C_1 C_2, Y_1 Y_2, M_1 M_2, and D_1 D_2 to facilitate further processing of the dates.

Ohms teaches reformatting into Lilian format and thereafter processing the date data in the database utilizing the Lilian format.

Therefore, Ohms does not teach the claimed invention as recited in claim 1 for purposes of 35 U.S.C. §102. In addition Ohms would not have made the claimed invention as recited in claim 1 obvious to a person of ordinary skill in the art at the time the invention was made, under 35 U.S.C. §103. As explained above, not only does Ohms not teach or suggest the claimed invention recited in claim 1, it clearly teaches away from virtually every step of the method of the claimed invention as recited in claim 1.

For like reasons, there is also no disclosure or suggestion in Ohms of:

sorting the symbolic representations of dates (claim 4); or *reformatting each symbolic representation of a date into the format C_1 C_2 Y_1 Y_2 M_1 M_2 D_1 D_2* (claim 5); or s*orting the symbolic representations of dates* using a numerical-order sort (claim 6); or *storing the symbolic representation of dates* and their associated information back into the database (claim 9).

There is also no teaching or suggestion in Ohms of:

The method of claim 9, including the additional step, after the step of reformatting, of

manipulating information in the database *having the reformatted date information therein* (claim 10).

In addition, there is no teaching or suggestion in Ohms of:

converting pre-existing date information having a different format into the format wherein M_1 M_2 is the numerical month designator, D_1 D_2 is the numerical day designator and Y_1 Y_2 is the numerical year designator (claim 7) or selecting Y_A Y_B *such that* Y_B *is 0* (zero) (claim 8).

Claims 2 and 3 of the patent are dependent on claim 1 and are patentable along with claim 1. *See*, *In re Fine*, 837 F.2d 1071, 5 U.S.P.Q.2d 1596 (Fed. Cir. 1988).

Similarly, there is no teaching or suggestion in Ohms of the claimed invention as recited in any of the claims 11-15 of the patent.

The '118 Patent

The '118 patent (" '118") was before the Examiner in the original prosecution of the Dickens patent being reexamined. The Examiner clearly fully considered '118 and determined it not to be a reference under §102 or §103. The Examiner was right, since '118 does not teach the claimed invention as recited in any of the claims of the patent, nor does it render the claimed invention obvious. In fact, like Ohms, it teaches away from the claimed invention.

Disclosed in '118 is a "system and method for *modifying and operating a computer system* to perform date operations on date fields having a two digit representation for the year without erroneously mistaking the years 2000 *et seq.* for the years 1900 *et seq.*" (Col. 1, lines 11-14)

As a solution to this problem, '118 proposes:

[i]n accordance with the present invention, the current date operation routines nested in the body of the application program would be replaced with a call to one of a plurality of subroutines stored externally from the existing application

program, *as opposed to the date operation routine being reprogrammed to perform the date operation in a new format.* The subroutines will be able to *accommodate the date format currently employed* by the application program, thus making it *unnecessary to convert all of the data fields in files containing data used in the application program over to the new data format.* (Col 4, lines 27-38).

As an example '118 describes a program that would "perform[] a date comparison to determine when loan payments are overdue ... ." (Col.4, lines 39-40) According to the '118 method, the:

program statements which performed the above functions would be modified to include program statements which did the following:

1. *Call the subroutine* which performs the date comparison *passing today's date, the date the next payment is due, and a three byte parameter*, the first byte of which identifies the format of today's date, the second byte of which identifies the format of the date next payment is due, and the third byte of which is left available for a *return code indicative of the result of the comparison*;

2. *If the result received from the subroutine indicates that the date next payment is due is greater than today's date, indicate that the account is okay.* (Col 4, lines 48-62)

In order to do this, '118 suggests that "for the subroutines to be able to accommodate different date formats, certain information, namely the current date, end of 100 year cycle, and two possible century values, must be determined and made available to the subroutines." (Col. 4, line 66 - Col. 5, line 3) In addition, '118 teaches that "*each subroutine that performs a date operation will include a call to another subroutine which can determine this information.*" (Col. 5, lines 3-5)

Further according to '118 "[t]he above-mentioned information will be used in the subroutine(s) to assign a century value to the two digit representation of the year of the dates to be operated on such that the subroutine can accurately perform its intended function." (Col. 5, lines 21-25) According to the '118 method this is performed using *a form of windowing* in which:

> [t]he current date is determined ... in a format which utilizes a four digit representation for the year. Initially, the current date is set to the operating system date in the format 00YYDDD ... by way of example ... 0094263 .... *The current date is then compared to the date the system was installed* with the [date conversion] modifications (modified system install date) ... which, for the sake of example, is 1994032 .... *If the YYDDD portion of the current date, 94263 is greater than or equal to the corresponding portion of this modified system install date, 94032 ... then the century of the current date is set to the century of the modified system install date ... ."* ...
>
> If the current date appeared *less than the modified system install date* ... in the 00YYDDD format ... then the current date century would be *set in the format CCYYDDD to the century value for the modified system install date plus one ... .*
>
> (Col 5, lines 31-57)

The '118 method then determines "the end of the 100 year cycle" according to "several parameters [which] may be specified." These "may include the number of years of future dating required (default is 10), ... and whether the end of the 100 year cycle is to be updated daily ... ." (Col. 6, lines 4-13) "If the cycle is to be updated daily, then the starting date is set to the current date ..., as determined above. ... Next, the end of the 100 year cycle is determined by adding the number of years future dating required to the starting date ... ." (Col 6, lines 17-22).

Further, explains '118:

The application program currently operating in a particular computer system may have a comparison of two date fields as part of its operation. If so, the source code which performs the comparison can be replaced with a *call to DS2000R1, the name given to an exemplary comparison subroutine* useful in practicing the present invention ... .

...

As illustrated in FIG. 8, the call DS2000R1 ... is inserted into the application program, and includes parameters P1, P2, and P3. P1 and P2 are the *date fields which are to be compared.* For example, P1 could be "DATE-NEXT-PAYMENT-DUE" and P2 might be "TODAY" as referenced in the above sample of modified source code. P3 is a three byte field in which the first two bytes define the type of date field P1 and P2, respectively. *The third byte is a return code which will be set to a value indicative of the result of the comparison.* (Col. 8, line 35 - Col. 9, line 53)

In summary, the teaching of '118 is to "[c]all the subroutine which performs the date [operation] passing [two dates] and the three byte parameter [including] a return code indicative of the result of the [operation]." Windowing occurs in the called subroutine in a manner other than that of the claimed invention. This "on call" or "on the fly" windowing of at most two date data entries at a time is not the subject matter of the claimed invention.

There is, therefore, no teaching or suggestion in '118 of:

> selecting a 10-decade window with a Y_A Y_B value for the first decade of the window, Y_A Y_B *being no later than the earliest Y_1 Y_2 year designator in the database*; … .

The '118 method selects a 10-decade window utilizing the "date the system was installed."

There is also, therefore, no teaching or suggestion in '118 of:

> *determining a century designator C_1 C_2 for each symbolic representation of a date in the database*, C_1 C_2 having … ; … .

The teaching of '118 is to determine a century designator for at most two date data representations being processed in a called subroutine at any given time.

There is also, therefore, no teaching or suggestion in '118 of:

> *reformatting the symbolic representation of the date with the values C_1 C_2, Y_1 Y_2, M_1 M_2, and D_1 D_2 to facilitate further processing of the dates.*

The teaching of '118 is to reformat two dates at a time in the called subroutine and the return to the program from the called subroutine an indicator of the result of the processing of the two reformatted date data entries. It does not teach facilitating "further processing of the dates" by "reformatting the symbolic representation of the date" "for each symbolic representation of a date in the database."

Accordingly, there is also no teaching or suggestion in '118 of:

sorting the symbolic representations of dates; (claim 4).

The method of '118 does not teach sorting all of the "symbolic representations of dates." It teaches only the comparison of two dates to each other in the called subroutine and returning to the program an indication of the result of the comparison.

There is also no teaching or suggestion in '118 of:

reformatting each symbolic representation of a date into the format $C_1\ C_2\ Y_1\ Y_2\ M_1\ M_2\ D_1\ D_2$ (claim 5), nor *sorting the symbolic representations of dates* using a numerical-order sort (claim 6); nor *storing the symbolic representation of dates* and their associated information back into the database (claim 9), nor manipulating information in the database *having the reformatted date information therein* (claim 10).

In addition, there is no teaching or suggestion in '118 of:

converting pre-existing date information[within a database] having a different format into the format wherein $M_1\ M_2$ is the numerical month designator, $D_1\ D_2$ is the numerical day designator and $Y_1\ Y_2$ is the numerical year designator (claim 7).

In addition, there is no teaching or suggestion in '118 of:

selecting $Y_A\ Y_B$ *such that* Y_B *is 0* (zero) (claim 8).

Claims 2 and 3 of the patent are dependent on claim 1 and are patentable along with claim 1. *In re Fine*, *supra*.

Similarly, there is no teaching or suggestion in '118 of the <u>claimed invention</u> as recited in any of the claims 11-15 of the patent.

Regarding the specific positions taken by the Commissioner, the patent owner responds as follows. The Commissioner has stated that one process step, "selecting Y_AY_B such that Y_B is 0 (zero)" (claims 8 and 13) is "taken as an arbitrary choice because there is no claim of use of this particular value, as there would be if the fast sort techniques noted in the specification were claimed." The patent owner respectfully disagrees that there is a need to expressly state this purpose for selecting Y_B to be zero in order for the claim to recite other than an "arbitrary choice." The Commissioner is correct in pointing out that the Specification (Col. 3, lines 13-23) explains the utility of this process step, i.e., the conversion need only utilize one digit for determining the century designator. The patent owner is unaware of any requirement that a process step recite, in addition to the step itself, the reason for performing the step or the outcome of the step, in order to recite something other than an "arbitrary choice." Indeed, there may be multiple reasons for the step, of which the specification may not need to list all. The recited process step is not arbitrary. It is performed for a reason that is explained in the Specification. The claim recitation, without more, is proper and not arbitrary.

The Commissioner states that Ohms discloses "A method of processing symbolic representations of dates stored in a database, comprising the steps of:" As noted above, at least in regard to the use of windowing, Ohms actually teaches a method of entering data into a database, rather than for processing symbolic representations in the database. The Commissioner seems to be saying that the first paragraph of claim 1 is also disclosed in Ohms. The patent owner agrees with what the Commissioner concludes, i.e., "the claim does not require a separate use of the digits." However, since Ohms does not disclose the step of the method of the <u>claimed invention</u>, as recited in the first paragraph of claim 1, as noted above, it is not seen how this comment of the Commissioner relates to the application of Ohms to the first paragraph of claim 1.

In regard to the second paragraph of claim 1, the Commissioner seems to find this step in the process of the claimed invention to be obvious from the disclosure of Ohms. However, what Ohms teaches is using a given year as "the starting point of a 100-year range" for purposes of windowing during input of date data into a database which is converted into Lilian format for storing and manipulation within the database. This does not, as noted above, teach or suggest the process step of the claimed invention, i.e., selection of "Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database." The Commissioner states, "[t]he skilled artisan of the time of the invention would have been motivated to select the pivot Y_AY_B less than or equal to Y_1Y_2 because it would separate two digit dates into the proper century in an efficient manner … ." This is speculation, without support in any cited reference. However, even so, Ohms' selection of a pivot is for purposes of windowing data entry and not what is recited in the claim. The earliest Y_1Y_2 in the database taught by Ohms could be the earliest Lilian date 0000001, October 15, 1582, and a ten decade window from there does not even get out of the sixteenth century. Ohms clearly teaches windowing for data entry only, along with storing and manipulating in Lilian format. The earliest date in the database is not relevant in Ohms' teaching of Ohms' use of windowing.

Similarly, as to the Commissioner's comments regarding the third paragraph of claim 1, the disclosure in Ohms to which the Commissioner refers relates to windowing for data input only and does not involve the process step recited in the claimed invention, i.e., "[d]etermining a century designator C_1C_2 for each symbolic representation of a date in a database … ." As noted above, the date data in Ohms' database is in Lilian format, and, according to his teaching, for that reason needs no century designators.

The patent owner does not fully understand the Commissioner's comments in the paragraph starting "The distinction between the earliest …", however, those comments do not negate the facts, as noted above, that the Ohms' method is clearly distinguishable from the method of the <u>claimed invention</u>. Also, the <u>claimed invention</u> does not recite "providing that Y_AY_B not be equal to the earliest Y_1Y_2" as the Commissioner states would be obvious. The Commissioner also fails to cite any reference to support this speculation.

The Commissioner fails to comment <u>at all</u> on the process step of the <u>claimed invention</u> as recited in the last paragraph of claim 1, and with good reason. The process step recited in this paragraph of the <u>claimed invention</u> is nowhere taught or suggested in Ohms. Ohms does not teach or suggest "reformatting the symbolic representations of the date with the values C_1C_2, Y_1Y_2 … ." Ohms teaches reformatting the dates in a database into the Lilian format.

In regard to claim 4 of the patent, the Commissioner asserts that "[a]lthough the sorting of the symbolic values is not specifically addressed in Ohms, it would have been readily apparent to sort the date values, because this makes it possible to efficiently prioritize records that include dates, such as orders to be filled." This is pure speculation without citation of any reference to suggest this process step, alone or in combination with other recited process steps. Even so, the failure of Ohms to disclose the basic process steps recited in claim 1, as noted above, renders this rejection of claim 4 untenable. Even if this process step were obvious, and the patent owner does not concede that it is, the combination of this step with the disclosure of Ohms does not result in the combination of process steps which constitutes the claimed invention.

In regard to the Commissioner's comments in respect of claims 5 and 6 of the patent, the patent owner is again at a loss to understand the relevance of the comments to the patentability of the claims. The comment that "Ohms converts to a Julian date with leading 4-digit year from a 2-digit year format," along with the relevance of the comment, is not fully understood. The fact remains, however, that such conversion by Ohms is in data input (or perhaps in data output to render the Lilian format date readable by humans in Julian format). It does not relate to the claimed invention as recited in claim 5 or claim 6. The Commissioner's comment that "[t]his places the century first and makes sorting more efficient, and in particular places the 20th century dates before the 21st century dates, as they should be," is also not understood, since Ohms does not teach or suggest doing this in any fashion relating to sorting or other data manipulation. Ohms sorts and otherwise manipulates the date data in Lilian format. Regardless of whether this "conversion would be equally applied to the Gregorian dates" the Ohms conversion for data input and utilization of Lilian format dates in the database is not the same as the method of the claimed invention.

The Commissioner again speculates without the citation of any reference, in regard to claim 7, that "[t]he skilled artisan would have been motivated to preconvert databases not in the format required ... in order to avoid the need for development of a separate system." The cited art does not teach or suggest this pre-conversion. Again, however, even were this an obvious process step, and the patent owner does not concede that it is, the combination of this step with Ohms still does not result in the process of the claimed invention as recited in claim 7.

The Commissioner suggests that the recitation of claim 8 "has no criticality in the claimed invention … ." As noted above, the patent owner respectfully submits that this is not correct, and the recitation of this process step without an additional recitation of why it is important and/or the beneficial result obtained by using this process step is entirely proper.

In regard to claim 9 of the patent, the Commissioner states that "[a]s to … storage [Ohms] states with respect to storage 'Of course, in the vast majority of cases, that is exactly what does take place.' " The portion of Ohms to which the Commissioner refers, however, is not referring to storage at all. Rather the portion of Ohms refers to the fact that, "[I]t is *not necessary to change date formats in files*, because it is *possible to change the programs only* so that the implied century in a date is recognized. Of course, in the vast majority of cases, that is exactly what does take place." What "that" refers to in this portion of Ohms is not entirely clear, but it is clear that it does not refer to the step of the claimed invention relating to "storing the symbolic representations of dates and their associated information back into the database."

In regard to claim 10 of the patent, the Commissioner again speculates, without the citation of a reference, that "[t]he skilled artisan would have been motivated to manipulate the information in the database, because the point of conversion is to provide for manipulating the information to avoid the Y2K problem." The fact remains, however, that Ohms does not teach or suggest "manipulating information in the database having the reformatted date information therein" as is recited in the claimed invention. Ohms manipulates, to the extent that manipulation is taught at all, in Lilian format not in the format recited in the claimed invention. The point of conversion in Ohms is to convert to Lilian format and not the conversion to the reformatted representations according to the claimed invention.

The Commissioner's discussion of '118 does not detract from the shortcomings of '118 as a reference, as noted above. The patent owner can generally agree that '118 "is generally directed to software patches of application programs [for example] for proper subtraction of dates." As noted above, the '118 patent discloses an "on-call" or "on-the-fly" type of software patch, a "call[ed] subroutine which performs the date comparison." In addition the patent owner agrees that '118 teaches utilization of "date pivots with a century span by using the current date ..., and install date ... ." There is also in '118 "a fairly broad discussion of date windowing within these constraints." These do not, however, as noted above, amount to a teaching or suggestion of the process of the claimed invention.

The Commissioner's brief comments that '118 "addresses the use of the earliest date in a **database**, through an *implicit assumption that this is related to the install date of an application program, and the (unspoken) possibility that such a program might be a data base management system (DBMS)*." This is speculation and supposition upon speculation and supposition, without the citation of any supporting reference. It is, further, unsupported by any teaching or suggestion in '118. The fact is that '118 makes no connection whatever between the install date and the earliest date in a database. It also makes no reference to database management systems in this regard. The discussion of '118 relating to "a fixed arbitrary date [being] entered" (Col. 6, lines 30-35) also has nothing whatever to do with selecting a date based upon the earliest date data in a database. The remainder of the discussion of '118 by the Commissioner is unrelated to the process steps as recited in the claimed invention. The patent owner respectfully submits that the discussion by the Commissioner does not dispel the patent owner's conclusions above in regard to the teachings and suggestions of '118.

As referenced in Appendix A to the Preliminary Amendment in the patent owner's Reissue Application, filed contemporaneously with this Response, Anonymous Submitter 3 also relies on Ohms and '118. The purported claim chart submitted to show that Ohms is a §102 reference as to claims 1-3, 7, 9 and 10 and a §103 reference in combination with Japan 05/027947 or "The Millennium Journal," each also referenced in that Appendix A, fails to apply Ohms to the claimed invention. The "conversion" referenced in the table on p. 247 of Ohms does not teach or suggest the process step of "providing a database with symbolic representations of dates stored therein according to a format wherein … ," as recited in the claimed invention. Neither does it teach or suggest "all of the symbolic representations of dates [in the database] falling within a 10 decade period of time." Ohms' selection of a desired starting date for data input windowing does not teach or suggest "selecting a 10-decade window … Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database." Ohms' determination of a century designator for data input is not the step of "determining a century designator C_1C_2 for each symbolic representation of a date in the database." Determining "the implied century" for data input is not "reformatting the symbolic representation of the date [in the database] with the values C_1C_2 … ." In regard to claim 7, Ohms does not teach or suggest the claimed conversion step. As to claim 9, Ohms' discussion of the "storage issue" has nothing to do with "storing the symbolic representations of dates and their associated information back in the database," as Ohms stores in Lilian format. For the same reason, the discussion of Ohms at p. 250 does not teach or suggest the claimed invention "manipulating information in the database having the reformatted date information therein."

Similarly, Anonymous Submitter 3 misses the mark regarding '118, said by Anonymous Submitter 3 to be a §102 reference for claims 1-7, 11 and 12 and a §103 reference as to claims 4, 6, 8 and 13 with Japan 05/027947. The fact that '118 may teach determining the "end of current 100 year cycle" does not teach or suggest "selecting ... Y_AY_B being no later than the earliest Y_1Y_2 year designator in the database," which '118 in fact does not teach or suggest. The fact that '118 utilizes windowing and that a century designator is determined by '118 for two selected dates, does not teach or suggest "determining a century designator C_1C_2 for each symbolic representation of a date in the database," which '118 does not anywhere teach or suggest. The fact that '118 does reformatting in the "[c]alled subroutine" for two date representations at a time does not teach or suggest "reformatting the symbolic representation of the date [each symbolic representation of a date in the database] ... to facilitate further processing of the dates [in the database]," which '118 does not teach or suggest.

In regard to claim 4, the fact that '118 teaches "date comparisons" between two dates in the "[c]alled subroutine" does not teach or suggest "sorting the [reformatted] symbolic representations of dates [in the database]," and, in the context of what '118 actually teaches, it is not true that "sorting is merely sequential date comparisons," as Anonymous Submitter 3 asserts. In regard to claim 6, the "dates to be operated on" referred to at Col. 6, lines 60-61 are the at most two dates passed to the "[c]alled subroutine," and are not "each symbolic representation of a date [in the database]." The same comments regarding claim 4 above are applicable to claim 6. The fact that '118 discloses "a host of formats" does not teach or suggest the step of the claimed invention found in claim 7. The above applies also to Anonymous Submitter 3's comments about claims 11 and 12.

In short, despite the Commissioner's comments, and those of Anonymous Submitter 3, neither Ohms nor '118 teach or suggest the claimed invention as recited in the claims of the patent, nor do they alone or together render obvious any of the claims. Neither Ohms nor '118 nor the combination of the two even presents a prima facie case for obviousness, since all of the elements of the claimed invention are not found in either reference or in the combination of the two. Indeed, if anything, they both teach away from the claimed inventions recited in the existing claims of the patent.

The patent owner asserts, therefore, that this Reexamination was improperly ordered since there is no substantial new question of patentability raised by either Ohms or '118 or the combination of the two. The patent owner respectfully requests that the Commissioner withdraw the Order or at a minimum stay this Reexamination pending the outcome of the patent owner's Reissue Application. During Reissue new claims of both broader and narrower scope than those that appear in the patent can be considered, and additional prior art references can be considered by the Office in light of the existing claims and the new claims. This will also avoid the patent owner having to pay the filing fee for a Preliminary Amendment in both this Reexamination Proceeding and in the Reissue Application.

In the event, however, that this Reexamination Proceeding is not dismissed or stayed, the patent owner reserves the right to submit a Preliminary Amendment in this Reexamination proceeding to place further claims before the Examiner which serve to even more particularly point out and distinctly claim the present invention and to define it over the prior art. In addition, in the event that this Reexamination is not withdrawn or stayed, the patent owner will submit an Information Disclosure Statement as submitted in the Reissue application.

Respectfully submitted,

[illegible]

William C. Cray
Levin & Haves
Registration No. 27,627
Attorneys for Dickens-Soeder2000

Telephone No. (949) 497-7676

90005592.021800

UNITED STATES PATENT AND TRADEMARK OFFICE

UNDER SECRETARY OF COMMERCE FOR INTELLECTUAL PROPERTY AND
DIRECTOR OF THE UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

DATE:	January 29, 2001
TO:	Pinchus Laufer; Vin Trans Special Program Examiners TC 2100
FROM:	Gerald A. Dost [illegible] Senior Legal Advisor Special Program Law Office
VIA:	Robert J. Spar [illegible] Director, Special Program Law Office Office of the Deputy Commissioner for Patent Examination Policy
SUBJECT:	First Office action now due in Dickens merged reissue/reexams.

The following reissue and three reexams have been filed on the Dickens 5,806,063 patent which relates to the turn of the century date handling in computer programs. The first reexam was a Commissioner ordered reexam and is very high profile.

1.	90/005,592	Commissioner ordered	Ordered on 12/21/99	Examiner Kulik
2.	90/005,628	Third party requester	Ordered on 03/10/00.	Examiner Kulik
3.	09/512,592	Reissue application	Filed on 02/23/00	Not assigned
4.	90/005,628	Third party requester	Ordered on 03/10/00	Examiner Homere

Over a year has passed since the Commissioner ordered reexam and the first Office action has not yet been issued. The delay to date can be justified, having been the result of the subsequent filings of the third party reexams and their statement and reply periods; the filing of the reissue by the patent owner; the merger of the reissue and the three reexams; and the required housekeeping amendment for entry in all four merged files.

The files are now ready for a first action on the merits. The action should be issued ASAP before any other action is taken by third parties and/or the patent owner which would further delay the proceedings. Please note that all four cases should be assigned to the same examiner.

cc: M. Focarino
J. Breene
K. Schor

UNITED STATES PATENT AND TRADEMARK OFFICE

UNDER SECRETARY OF COMMERCE FOR INTELLECTUAL PROPERTY AND
DIRECTOR OF THE UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

DATE:	January 29, 2001
TO:	Pinchus Laufer; Vin Trans Special Program Examiners TC 2100
FROM:	Gerald A. Dost [illegible] Senior Legal Advisor Special Program Law Office
VIA:	Robert J. Spar [illegible] Director, Special Program Law Office Office of the Deputy Commissioner for Patent Examination Policy
SUBJECT:	First Office action now due in Dickens merged reissue/reexams.

The following reissue and three reexams have been filed on the Dickens 5,806,063 patent which relates to the turn of the century date handling in computer programs. The first reexam was a Commissioner ordered reexam and is <u>very high profile</u>.

1.	90/005,592	Commissioner ordered	Ordered on 12/21/99	Examiner Kulik
2.	90/005,628	Third party requester	Ordered on 03/10/00.	Examiner Kulik
3.	09/512,592	Reissue application	Filed on 02/23/00	Not assigned
4.	90/005,628	Third party requester	Ordered on 03/10/00	Examiner Homere

Over a year has passed since the Commissioner ordered reexam and the first Office action has <u>not</u> yet been issued. The delay to date can be justified, having been the result of the subsequent filings of the third party reexams and their statement and reply periods; the filing of the reissue by the patent owner; the merger of the reissue and the three reexams; and the required housekeeping amendment for entry in all four merged files.

The files are now ready for a first action on the merits. The action should be issued ASAP before any other action is taken by third parties and/or the patent owner which would further delay the proceedings. Please note that all four cases should be assigned to the same examiner.

cc: M. Focarino
J. Breene
K. Schor

UNITED STATES PATENT AND TRADEMARK OFFICE

UNDER SECRETARY OF COMMERCE FOR INTELLECTUAL PROPERTY AND
DIRECTOR OF THE UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

DATE:	January 29, 2001
TO:	Pinchus Laufer; Vin Trans Special Program Examiners TC 2100
FROM:	Gerald A. Dost [illegible] Senior Legal Advisor Special Program Law Office
VIA:	Robert J. Spar [illegible] Director, Special Program Law Office Office of the Deputy Commissioner for Patent Examination Policy
SUBJECT:	First Office action now due in Dickens merged reissue/reexams.

The following reissue and three reexams have been filed on the Dickens 5,806,063 patent which relates to the turn of the century date handling in computer programs. The first reexam was a Commissioner ordered reexam and is <u>very high profile</u>.

1.	90/005,592	Commissioner ordered	Ordered on 12/21/99	Examiner Kulik
2.	90/005,628	Third party requester	Ordered on 03/10/00.	Examiner Kulik
3.	09/512,592	Reissue application	Filed on 02/23/00	Not assigned
4.	90/005,628	Third party requester	Ordered on 03/10/00	Examiner Homere

Over a year has passed since the Commissioner ordered reexam and the first Office action has <u>not</u> yet been issued. The delay to date can be justified, having been the result of the subsequent filings of the third party reexams and their statement and reply periods; the filing of the reissue by the patent owner; the merger of the reissue and the three reexams; and the required housekeeping amendment for entry in all four merged files.

The files are now ready for a first action on the merits. The action should be issued ASAP before any other action is taken by third parties and/or the patent owner which would further delay the proceedings. Please note that all four cases should be assigned to the same examiner.

cc: M. Focarino
J. Breene
K. Schor

UNITED STATES PATENT AND TRADEMARK OFFICE

UNDER SECRETARY OF COMMERCE FOR INTELLECTUAL PROPERTY AND
DIRECTOR OF THE UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

DATE:	January 29, 2001
TO:	Pinchus Laufer; Vin Trans Special Program Examiners TC 2100
FROM:	Gerald A. Dost [illegible] Senior Legal Advisor Special Program Law Office
VIA:	Robert J. Spar [illegible] Director, Special Program Law Office Office of the Deputy Commissioner for Patent Examination Policy
SUBJECT:	First Office action now due in Dickens merged reissue/reexams.

The following reissue and three reexams have been filed on the Dickens 5,806,063 patent which relates to the turn of the century date handling in computer programs. The first reexam was a Commissioner ordered reexam and is <u>very high profile</u>.

1.	90/005,592	Commissioner ordered	Ordered on 12/21/99	Examiner Kulik
2.	90/005,628	Third party requester	Ordered on 03/10/00.	Examiner Kulik
3.	09/512,592	Reissue application	Filed on 02/23/00	Not assigned
4.	90/005,628	Third party requester	Ordered on 03/10/00	Examiner Homere

Over a year has passed since the Commissioner ordered reexam and the first Office action has <u>not</u> yet been issued. The delay to date can be justified, having been the result of the subsequent filings of the third party reexams and their statement and reply periods; the filing of the reissue by the patent owner; the merger of the reissue and the three reexams; and the required housekeeping amendment for entry in all four merged files.

The files are now ready for a first action on the merits. The action should be issued ASAP before any other action is taken by third parties and/or the patent owner which would further delay the proceedings. Please note that all four cases should be assigned to the same examiner.

cc: M. Focarino
J. Breene
K. Schor

UNITED STATES PATENT AND TRADEMARK OFFICE

UNDER SECRETARY OF COMMERCE FOR INTELLECTUAL PROPERTY AND
DIRECTOR OF THE UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

DATE: January 29, 2001

TO: Pinchus Laufer; Vin Trans
Special Program Examiners
TC 2100

FROM: Gerald A. Dost [illegible]
Senior Legal Advisor
Special Program Law Office

VIA: Robert J. Spar [illegible]
Director, Special Program Law Office
Office of the Deputy Commissioner
for Patent Examination Policy

SUBJECT: First Office action now due in Dickens merged reissue/reexams.

The following reissue and three reexams have been filed on the Dickens 5,806,063 patent which relates to the turn of the century date handling in computer programs. The first reexam was a Commissioner ordered reexam and is very high profile.

1.	90/005,592	Commissioner ordered	Ordered on 12/21/99	Examiner Kulik
2.	90/005,628	Third party requester	Ordered on 03/10/00.	Examiner Kulik
3.	09/512,592	Reissue application	Filed on 02/23/00	Not assigned
4.	90/005,628	Third party requester	Ordered on 03/10/00	Examiner Homere

Over a year has passed since the Commissioner ordered reexam and the first Office action has not yet been issued. The delay to date can be justified, having been the result of the subsequent filings of the third party reexams and their statement and reply periods; the filing of the reissue by the patent owner; the merger of the reissue and the three reexams; and the required housekeeping amendment for entry in all four merged files.

The files are now ready for a first action on the merits. The action should be issued ASAP before any other action is taken by third parties and/or the patent owner which would further delay the proceedings. Please note that all four cases should be assigned to the same examiner.

cc: M. Focarino
J. Breene
K. Schor

01-08-01 2177

09/512592

LEVIN & HAWES LLP
ATTORNEYS AT LAW INTELLECTUAL PROPERTY LAW

~~384~~ FOREST AVENUE SUITE 13
LAGUNA BEACH, CALIFORNIA 92651
TEL 949.497.7676
FAX 949.497.7679
www.lagunalaw.com

OIPE JC127 JAN 05 2001 PATENT & TRADEMARK OFFICE

RECEIVED JAN 09 2001 Technology Center 2100

Patent ◆ Trademark ◆ Copyright ◆
Trade Dress ◆ and Related Litigation

Via: Express Mail ET051659769US

January 5, 200~~0~~1
Box: Non-Fee Amendment
Assistant Commissioner for Patents
Washington, DC 20231

COPY

RECEIVED
JAN 16 2001
OFFICE OF PETITIONS

Dear Commissioner:

Enclosed is a **Housekeeping Amendment** in the merged cases:

✓ **Reissue Application No.: 09/512,592**)	**Group Art Unit:** 2177
United States Patent No.: 5,806,063)	**Examiner:** Paul Kulik
Issued: September 8, 1998)	**Attorney Docket No.:** 2039-154
Applicant: Dickens-Soeder2000,LLC)	
Reexamination Proceeding: 90/005,592)	
Filed: December 21, 1999)	
Reexamination Proceeding: 90/005,628)	
Filed: February 2, 2000)	
Reexamination Proceeding: 90/005,727)	
Filed: May 16, 2000)	

Attorney Docket No.: 2039-154

This Amendment consists of:
Housekeeping Amendment of 36 pages
New Claims 16-76
Information Disclosure Statement
USPTO Form PTO/SB/08A
Supplemental Information Disclosure Statement Submission
Certificate of Mailing
Certificate of Service By Mail
Return Receipt Postcard

OIPE JC127
JAN 0 5 2001
PATENT & TRADEMARK OFFICE

The fee and certification requirements of 37 C.F.R. §1.97 have been waived pursuant to the DECISION, *SUA SPONTE*, TO MERGE REEXAMINATION AND REISSUE PROCEEDINGS, mailed November 6, 2000. The fee for examination of claims in excess of the original filing fee have been paid in the above referenced Reissue Application.

If you have any questions, please do not hesitate to contact me.

Regards,

[illegible]

William C. Cray
Registration No. 27627

RECEIVED
JAN 1 6 2001
OFFICE OF PETITIONS

WCC/ns
Enclosures

OIPE JC127
JAN 0 5 2001
PATENT & TRADEMARK OFFICE

RECEIVED
JAN 0 9 2001
Technology Center 2100

January 5, 2001

BOX: NON-FEE AMENDMENT
Assistant Commissioner for Patents
Washington, DC 20231

CERTIFICATE OF MAILING UNDER 37 CFR § 1.10

Re: **Housekeeping Amendment** in the merged cases:

Reissue Application No.:)	**Group Art Unit:** 2177	
09/512,592)		
United States Patent No.:)	**Examiner:** Paul Kulik	RECEIVED
5,806,063)		
Issued: September 8, 1998)	**Attorney Docket No.:**	JAN 1 6 2001
Applicant:)	2039-154	
Dickens-Soeder2000,LLC)		OFFICE OF PETITIONS
Reexamination Proceeding:)		
90/005,592)		
Filed: December 21, 1999)		
Reexamination Proceeding:)		
90/005,628)		
Filed: February 2, 2000)		
Reexamination Proceeding:)		
90/005,727)		
Filed: May 16, 2000)		

Attorney Docket No.: 2039-154

Enclosed with this Certificate of Mailing is:
Housekeeping Amendment of 36 pages
New Claims 16-76
Information Disclosure Statement
USPTO Form PTO/SB/08A
Supplemental Information Disclosure Statement Submission
Certificate of Service By Mail
Return Receipt Postcard

The fee and certification requirements of 37 C.F.R. §1.97 have been waived pursuant to the DECISION, *SUA SPONTE*, TO MERGE REEXAMINATION AND REISSUE PROCEEDINGS, mailed November 6, 2000. The fee for examination of claims in excess of the original filing fee has been paid in the above referenced Reissue Application.

[illegible]

Nanees Salama

CHAPTER 7

The Preprocessor

When we discussed the Clipper 5 compilation process in Chapter 1, we introduced the preprocessor. If the new compiler is the heart of the Clipper 5 package, then the preprocessor must be the soul. They work hand in hand to make Clipper the completely open-ended language that it has become.

After reading this chapter, you will use the preprocessor to make your programs run faster, be easier to read, and leap tall mainframes at a single bound (if not one, then certainly two). We will demonstrate many ways to use preprocessor directives like manifest constants and user-defined commands to your best advantage.

Overview

C programmers already know the benefits that a preprocessor offers. Now Clipper developers can share in those benefits, because the Clipper 5 release includes its own powerful built-in preprocessor. This feature alone speaks volumes for Nantucket's conviction in making Clipper 5 a bona fide development language.

A preprocessor looks at your source code and performs certain translations prior to the main compilation step. Based upon its limitless capacity for translation, some Clipper developers have gone so far as to describe the preprocessor as a glorified search-and-replace mechanism. This is in effect like piloting an airplane and never getting it off the tarmac, because the preprocessor does a lot more than search and replace. Its operations include:

- translation (simple and complex)
- inclusion of other files (known in Clipper 5 as "header files" and denoted by the file extension .CH)
- conditional compilation of certain blocks of code (which is great for debugging and/or demonstration versions)

Because the preprocessor is integrated into the Clipper 5 compiler, you are going to be using it whether you know (or like) it or not, so you might as well learn how to make it work to your benefit. In fact, the issue of upward compatibility (making Summer '87 code run under Clipper 5) is neatly addressed by the preprocessor with the STD.CH file. (We will talk more about this file later.)

Manifest constants

Manifest constants are identifiers that the preprocessor acts upon. They have many uses, including improved readability, faster execution speed, and conditional compilation.

Improving readability

With the preprocessor you will now be able to substitute meaningful names instead of cryptic numbers. The **#define** directive allows you to declare **manifest constants.** The syntax for defining a manifest constant is:

```
#define <identifier> [<value>]
```

The preprocessor will sniff out all occurrences of <identifier> in your source code, and replace them with <value> (hence the aforementioned search-and-replace comparison).

Be aware that the preprocessor's substitution for manifest constants is **case-sensitive**. You may wish to avoid problems by adhering to the C naming convention of all uppercase for manifest constants. This is the convention that we will use throughout this book. (Note that you can also define manifest constants without using the <value> parameter. We will discuss this situation in a moment.)

Before we delve into source code examples, be warned that you will see many examples of original source code and preprocessed output with the latter following hot on the heels of the former (and not just in this chapter either). Rest easy — the code that you write is the original source code. The preprocessed output is what it looks like after the preprocessor has finished mangling it. Although you do not have to look at the preprocessed output (let alone do anything with it), we nonetheless recommend that for your first few months with Clipper 5, you make heavy use of the /P compiler option (previously discussed in Chapter 1). This option will create a preprocessed output file, which will have the same filename as your PRG but with the extension PPO. Carefully scrutinize the PPO file to see what the preprocessor is doing to your source code.

Okay, on with the first source code example:

Original source code (.PRG):

```
#define K_DOWN  24
#define K_UP     5
#define K_LEFT  19
#define K_RIGHT  4

if keypress == K_DOWN .or. keypress == K_UP   .or. ;
   keypress == K_LEFT .or. keypress == K_RIGHT
```

Preprocessed output (.PPO):

```
if keypress == 24 .or. keypress == 5 .or. keypress == 19 ;
     .or. keypress == 4
```

(Note that the preprocessor strips out whatever is not translated and leaves blank lines in its wake. There would thus be several blank lines at the top of the PPO file.)

As you can see, it is far more obvious what is happening when you use words rather than numbers. You may have an infinite capacity for memorizing numbers and can recite the entire INKEY() value list in your sleep. But using words in your source code will help the poor soul who will inherit your code next year and be forced to maintain it.

Simply put, we humanoids prefer words and computers prefer numbers, and with the preprocessor there is less need for us to waste time looking up INKEY() values. In fact, Nantucket provides a header file, INKEY.CH, which contains manifest constants for all of the INKEY() equivalents. Because they use a standard naming convention (K_ENTER, K_TAB, K_CTRL_E, and so on), you will probably never need to look at this file either. Just include it in your programs, and leave the drudgery to the preprocessor.

Another reason to use manifest constants is that the preprocessor will allow you to use as many as 32 characters in a manifest constant as compared to the customary 10 characters in a variable name. You can say a lot more in 32 characters than 10!

Arrays vs. memvars

Although we have not covered arrays yet, you will learn in Chapter 9 how to save memory by using arrays rather than **private** or **public** memory variables to hold field values while editing. This is because you can cut down on the number of symbols in your program, and thus shrink your symbol table (which we discussed at length in Chapter 6, "Variable Scoping").

There is only one drawback to using arrays instead of memory variables — it often leads to unreadable code. But with the preprocessor, we get the best of both worlds: We can use the array but establish manifest constants to give each array element more obvious meaning.

```
local aMemvars[8]

#define MFNAME      aMemvars[1]
#define MLNAME      aMemvars[2]
#define MADDRESS    aMemvars[3]
#define MCITY       aMemvars[4]
#define MSTATE      aMemvars[5]
#define MZIP        aMemvars[6]
#define MFRIEND     aMemvars[7]
#define MBIRTHDATE  aMemvars[8]
```

Then you can write your GETs and actually be able to figure out what is happening:

```
@ 7, 28 get MFNAME
@ 8, 28 get MLNAME
@ 9, 28 get MADDRESS
@ 10, 28 get MCITY
@ 11, 28 get MSTATE
@ 12, 28 get MZIP
@ 13, 28 get MFRIEND picture "Y"
@ 14, 28 get MBIRTHDATE
```

whereas the array equivalent is not going to be a very pretty sight:

```
@ 7, 28 get aMemvars[1]
@ 8, 28 get aMemvars[2]
@ 9, 28 get aMemvars[3]
@ 10, 28 get aMemvars[4]
@ 11, 28 get aMemvars[5]
@ 12, 28 get aMemvars[6]
@ 13, 28 get aMemvars[7] picture "Y"
@ 14, 28 get aMemvars[8]
```

Improving execution speed

Let's review our first example in this chapter, the keypress test. Another way that we could have approached the problem would be to define variables instead of manifest constants:

```
K_DOWN  := 24
K_UP    :=  5
K_LEFT  := 19
K_RIGHT :=  4
```

In prior versions of Clipper, this was the only method available. Many developers used such "pseudo-manifest constants" to improve the readability of their programs. But there are two drawbacks to this method as compared to true preprocessed manifest constants:

- These pseudo-constants are held in the symbol table rather than being resolved at compile-time. This means that whenever they are referred to at run-time, their value must be looked up in the symbol table. Although this does not take a lot of time, it does slow things down by a few clock cycles. Let's do a simple timing test:

  ```
  * using pseudo-constants
  TEST := 5
  for xx := 1 to 1000
     y := TEST
  next

  * using manifest constants
  #define TEST 5
  for xx := 1 to 1000
     y := TEST
  next
  ```

  The second loop runs approximately 10% faster. Another advantage is that the size of your program will be smaller because no symbol table entry is required for the manifest constant (as would be for a traditional memory variable).

- Pseudo-constants are always subject to accidental change during the course of your program. For example, what is to keep you from switching between numeric and character type?

```
* at the top of the program
K_RIGHT := 4

* 3000 lines further down!
K_RIGHT := chr(4)
```

What would happen the next time your program refers to K_RIGHT? Flip a coin and find out.

You may protest that it is unlikely you would make such a silly mistake. We agree, you are an excellent programmer. However, there is always the possibility that some other programmer will make future modifications to your source code, and they might not be as perfect as you!

While we're on the issue of speed, you can also save time by substituting one-line function calls with preprocessor macros. These are definitely not the same (nor anywhere close) to the traditional dBASE macro (i.e., &something). For the sake of clarity, we will refer to preprocessor macros as **pseudo-functions**. The syntax for defining a pseudo-function is quite similar to that of a manifest constant:

```
#define <function>([<argument list>]) <expression>
```

The preprocessor will track down all occurrences of <function> in your source code, and replace them with <expression>. If you specify an optional <argument list>, those arguments will be substituted into the <expression> based on the names you give them in the <argument list>. For example:

```
#define WHATEVER(exp1, exp2)   exp1 + exp2
x := WHATEVER("ABC", "123")
```

will be preprocessed into:

```
x := "ABC" + "123"
```

Because we specified an argument list (**exp1** and **exp2**), **exp1** assumed the valu "ABC", and **exp2** the value of "123". The preprocessor was then kind enough t substitute them in the expression **exp1 + exp2**.

If you choose to specify an argument list, you must follow some simple rules:

• No whitespace between the function name and the open parenthesis.

```
#define WHATEVER(exp1, exp2)   exp1 + exp2      // fine
#define WHATEVER (exp1, exp2)   exp1 + exp2     // nope
```

• A closing parenthesis must follow the argument list.

Let's write a pseudo-function Maxx(), which will accept three numeric parameters, and return the maximum of the three. First we will write it as a regular UDF:

```
x1 := 500
x2 := 499
for xx := -1000 to 1000
   yy := Maxx(x1, xx, x2)
next

function Maxx(a, b, c)
return max(if(a > b, a, b), c)
```

Now we will try it again as a pseudo-function:

Original (.PRG):

```
#define MAXY(a, b, c)  max(if(a > b, a, b), c)
x1 := 500
x2 := 499
```

```
for xx := -1000 to 1000
   yy := MAXY(x1, xx, x2)  // note uppercase
next
```

Preprocessed output (.PPO):

```
x1 := 500
x2 := 499
for xx := -1000 to 1000
   yy := max(if(x1 > xx, x1, xx), x2)
next
```

The pseudo-function runs 25% faster than the function call (and that is even when we use **local** parameters in the function — if you stick to **privates**, the relative speed of that function call will seem even more miserable). The reason for this performance gain is because a function call adds a certain amount of overhead, however slight. Having everything in-line means that the computer does not have to jump to a different function and set everything up to return the appropriate value and then jump back. Where performance is of paramount concern, having in-line code can remove the bottlenecks at run-time. Granted, these "bottlenecks" are not nearly as bad as the traffic on your average Southern California freeway at rush hour, but every little bit helps.

Pseudo-functions also give us plenty of rope with which to hang ourselves. Whenever you expand an argument list, you should be very careful of parenthetical grouping. Consider Times(), a simple pseudo-function which accepts two numerics and multiplies them together.

Original (.PRG):

```
#define TIMES(a, b)   a * b
w := 5
x := 4
y := 3
z := 2
t := TIMES(w + x, y + z)
```

Preprocessed output (.PPO):

```
t := w + x * y + z
```

As you may recall from our discussion of operator precedence in Chapter 5, multiplication will occur before addition. This outcome is not what we had in mind. Rather than 45 ((5 + 4)(3 + 2)), the variable **t** will be assigned the value of 19 (5 + (4 * 3) + 2). You are the one who left out the parentheses; the preprocessor was merely following orders. Let's fix it now!

```
#define Times(a, b) (a) * (b)
```

Demos and debugging

Have you ever included debugging code in your program?

```
// MYPROG.PRG
debug := .T.
*
* elsewhere in the program
if debug
   ? "procname() = ", procname()
   ? "procline() = ", procline()
   ? "readvar() =  ", readvar()
   ? "memory(0) =  ", memory(0)
   ? "x = ", x
   ? "y = ", y
   ? "z = ", z
endif
```

In the same manner, have you ever included code in your programs so that you could distribute a demonstration version to prospective clients?

```
if demo
   ? "This demo will access only 50 records"
   max_rec := 50
else
   max_rec := 5000000000   // mammoth file
endif
```

Even though these blocks of code will only be executed conditionally, ALL of the source code will be compiled unconditionally. Therefore, it will all end up in your object modules, and consequently, your EXE file. This is a senseless waste of memory.

Fortunately, the preprocessor provides us with the power to conditionally compile our source code. Earlier, we mentioned that you could define manifest constants without a <value> parameter, and this is exactly such an instance. By its presence alone, this type of manifest constant directs the preprocessor to compile (or not compile) sections of source code. The syntax is:

```
#define <identifier>
```

This <identifier> does not require a value. All it needs to do is exist. However, it will not be truly useful until you make use of the **#ifdef** and **#ifndef** directives. #ifdef tells the preprocessor that if a certain identifier exists, it should compile the following block. #ifndef does the opposite; it directs the preprocessor to compile the following block only if the identifier does not exist.

Let's tackle that debug example again with these new directives:

Original (.PRG):

```
#define DEBUG
#ifdef DEBUG
   ? "procname() = ", procname()
   ? "procline() = ", procline()
   ? "readvar() =  ", readvar()
   ? "memory(0) =  ", memory(0)
   ? "x = ", x
   ? "y = ", y
   ? "z = ", z
#endif
```

Preprocessed output (.PPO):

```
qout("procname() = ", procname())
qout("procline() = ", procline())
qout("readvar() =  ", readvar())
qout("memory(0) =  ", memory(0))
qout("x = ", x)
qout("y = ", y)
qout("z = ", z)
```

If we do not #define the DEBUG manifest constant, watch what happens:

Original (.PRG):

```
#ifdef DEBUG
   ? "procname() = ", procname()
   ? "procline() = ", procline()
   ? "readvar() =  ", readvar()
   ? "memory(0) =  ", memory(0)
   ? "x = ", x
   ? "y = ", y
   ? "z = ", z
#endif
```

Preprocessed output (.PPO):

```
(Space, the final frontier.)
```

Now you can safely leave in all your debugging code without fear of bulking up your EXE file. Just #define DEBUG when (or should we say if) you need to use it again!

Because you are a smart cookie, you have probably already surmised that, where there's an IF and an ENDIF, there is probably an ELSE. Sure enough, we'll use it to clean up that demo example:

Original (.PRG):

```
#define DEMO
#ifdef DEMO
   ? "This demo will access only  50 records"
   max_rec := 50
#else
   max_rec := 5000000000   // mammoth file
#endif
```

Preprocessed output(.PPO):

```
qout("This demo will access only 50 records")
max_rec := 50
```

Get rid of the DEMO definition, and...

Original (.PRG):

```
#ifdef DEMO
   ? "This demo will access only 50 records"
   max_rec := 50
#else
   max_rec := 5000000000
#endif
```

Preprocessed output (.PPO):

```
max_rec := 5000000000
```

In similar fashion, #ifndef allows conditional compilation based on the non-existence of a specific identifier:

Original (.PRG):

```
#ifndef REALTHING
   ? "This demo will access only 50 records"
   max_rec := 50
#else
   max_rec := 5000000000
#endif
```

Preprocessed output (.PPO):

```
qout("This demo will access only 50 records")
max_rec := 50
```

There is one more directive in this collection that may serve you well. **#undef** removes (undefines) an identifier. This has several purposes, the first being to restrict conditional compilation to a section of your program:

Original (.PRG):

```
#define DEMO
#ifdef DEMO
   max_rec := 50
   max_calls := 100
#else
   max_rec := 50000000
   max_calls := 100000
#endif

#undef DEMO

#ifdef DEMO
   max_times := 25
#else
   max_times := 200
#endif
```

Preprocessed output (.PPO) (most blank lines omitted):

```
max_rec := 50
max_calls := 100

max_times := 200
```

Notice what happened in the bottom #ifdef..#else..#endif block because you undefined the DEMO identifier. Because you #undefined the DEMO identifier, the preprocessor conditionally compiled as if you were not using the demo code.

Another instance would be where you would want to redefine a manifest constant. This will generate a compiler warning, unless you undefine it first:

```
#define DEMO
#ifdef DEMO
   max_recs := 50
#else
```

```
   max_recs := 10000
#endif

#undef DEMO        // remove this and watch the compiler whine!

#define DEMO .t.
```

When you #define a manifest constant, it is visible from that line until either the end of that PRG file or you #undefine it. This rule also applies to manifest constants in header files that you #include (which we will cover in greater detail very soon).

The only exception to this rule are manifest constants that have been #defined in the STD.CH header file, or an alternate standard rules file that you specify with the /U compiler option. The reason that these manifest constants are visible everywhere is because they are loaded as soon as you fire up the compiler (CLIPPER.EXE), and therefore have a scope that is not limited to any particular .PRG file.

The /D compile option

As we mentioned in Chapter 1, you can #define manifest constants at compile-time with the exceedingly clever /D compiler option. This allows you to change manifest constants without ever touching your source code. You can either create new manifest constants, or, with careful use of the #ifndef directive, override existing ones.

The syntax for the /D compiler option is:

```
clipper progname /D<ID>[=<VAL>]
```

<ID> represents the name of the identifier. You may optionally assign a value <VAL> to the identifier by following <ID> with an equal sign and the value. For example, in the last code fragment we could have removed the #define DEMO statement and compiled the program like so:

```
clipper test /dDEMO
```

This would have exactly the same effect.

You may optionally assign a value to an identifier. Suppose that you want to initialize an array to a certain size. Other things, such as FOR..NEXT loops, also rely upon the array size, and you want to be able to change all such references in one fell swoop. The easiest way to do this would be to define an identifier (or **manifest constant**) at the top of your program like so:

```
#define ELEMENTS  500
local a[ELEMENTS], total
for x := 1 to ELEMENTS
   total += (a[x] := x)
next
```

Now suppose that you want to be able to change the number of elements without changing your source code. With the /D switch, it's easy. Compiling your program with the following command-line arguments will result in an array (and loop counter) of 1000 rather than 500.

```
clipper myprog /dELEMENTS=1000
```

If you have jumped the gun and compiled this example already, you should now be complaining about a "redefinition of #define" compiler error. That makes sense — you are defining ELEMENTS twice: once when you fire up the compiler, and again within the source code where you originally #defined it. Here's a simple solution to this problem:

```
#ifndef ELEMENTS
   #define ELEMENTS  500
#endif

local a[ELEMENTS], total
for x := 1 to ELEMENTS
   total += (a[x] := x)
next
```

This tells the preprocessor to define ELEMENTS only if it has not already been defined.

The more you use manifest constants, the more time you will be able to save with the /D compile option.

Header files

Now that you are ready to build an impressive collection of your own manifest constants, you should know how to segregate them from your source code. **Header files** (also known as "include files") are the best repositories for manifest constants and user-defined commands. Clipper 5 header files generally carry the extension of ".CH".

For example, instead of putting them at the top of every source code file that uses them, like this:

```
#define CRLF            chr(13)+chr(10)
#define MAXY(a, b, c)   max(if(a > b, a, b), c)
#define NETERR_MSG      "Network error, could not add/edit" +;
                        "at this time"
#translate CENTER( <row>, <msg>) => ;
   DevPos( <row>, int((maxcol()+1 - len( <msg> )) / 2)) ;;
   DevOut( <msg> )
```

you can put them into a .CH file and then simply #include that in your programs:

```
#include "MYSTUFF.CH"
```

The #include directive is self-explanatory: It includes the contents of a header file at compile-time. You must always surround the name of the header file with quotes, and the extension must be specified. You may also optionally specify a drive and path, but if you do not, the preprocessor will search in the following three places (in order):

- The current directory.
- Directories specified by the "/i" compiler option (see Chapter 1).
- Directories listed in the INCLUDE environmental variable. (As discussed in Chapter 1, the recommended setting is SET INCLUDE=C:\CLIPPER5\INCLUDE.)

Although header files will usually contain manifest constants and user-defined commands, they may also include regular source code (except for STD.CH and any alternate standard rules file).

However, we do not necessarily advocate doing this because it will make source level debugging difficult, if not totally impossible. Another reason against putting source code in header files is that it is contrary to the purpose of such files, which exist primarily to contain preprocessor directives.

#including a header file is not the same thing as calling another program file with the DO command! When you #include a header file, the preprocessor will use only what it needs. As a result, the size of your compiled code will be kept to an absolute minimum.

You may nest #include directives up to sixteen levels deep:

```
* FILE1.PRG
#include "FILE2.CH"
*

* FILE2.CH
#include "FILE3.CH"
*

* FILE3.CH
#include "FILE4.CH"
```

and so on.

Nantucket header files

The following files are supplied by Nantucket with Clipper 5. These should all be in your \CLIPPER5\INCLUDE directory. Most of these header files contain manifest constants which follow a standard naming convention to make them easier to remember.

Table 7.1 Nantucket header files

Filename	*Relevant to*	*Prefix*
ACHOICE.CH	ACHOICE() user-defined function	AC_
BOX.CH	Box-drawing commands	B_
DBEDIT.CH	DBEDIT() user-defined function	DE_
DBSTRUCT.CH	DBSTRUCT()	DBS_
DIRECTRY.CH	DIRECTORY()	F_
ERROR.CH	Clipper 5 error codes	EG_
FILEIO.CH	Low-level file functions	F_, FS_, FO_, FC_
GETEXIT.CH	get:exitState values	GE_
INKEY.CH	INKEY() values (very handy)	K_
MEMOEDIT.CH	MEMOEDIT() user-defined function	ME_
RESERVED.CH	To resolve naming conflicts	n/a
SET.CH	SET()	_SET_
SETCURS.CH	SETCURSOR()	SC_
SIMPLEIO.CH	Simplified input/output commands	n/a
STD.CH	Standard Clipper 5 language definition	n/a

Rather than list the contents of each of these files line by line, we will present a self-explanatory example that uses several of them.

Listing 7.1 Manifest constant/#include example

```
#include "BOX.CH"
#include "INKEY.CH"
#include "SET.CH"
#include "SETCURS.CH"
#define OFF .f.
#translate CENTER( <row>, <msg> ) => ;
 DevPos( <row>, int((maxcol() + 1 - len( <msg> )) / 2)) ; ;
 DevOut( <msg> )

function main
local key, oldcursor := setcursor(SC_NONE)  // turn off cursor
set(_SET_SCOREBOARD, OFF)
set(_SET_CANCEL, OFF)
cls
@ 0, 0, 24, 79 box B_DOUBLE + ' ' color 'w/b'
@ 6, 6, 18, 73 box B_SINGLE + ' ' color 'w/r'
@ 11, 18, 13, 61 box B_SINGLE_DOUBLE + ' ' color '+w/rb'
do while .t.
   center(12, "Press a key - Esc to exit")
   key := inkey(0)
   scroll(12, 19, 12, 60, 0)
   do case
        case key == K_ENTER
           center(12, "You pressed Enter")
        case key == K_F1
           center(12, "No help available")
        case key == K_SH_F1
           center(12, "Still no help available")
        case key == K_ALT_A
           center(12, "You pressed Alt-A")
        case key == K_CTRL_Y
           center(12, "You pressed Ctrl-Y")
        case key == K_ESC
           exit
        otherwise
           center(12, "Unknown keypress")
   endcase
   inkey(1)
enddo
setcursor(oldcursor) // restore cursor
return nil
```

User-defined commands

There are several drawbacks to using #define to create preprocessor functions. The biggest is that, as mentioned above, the preprocessor treats #define directives as case-sensitive. This means that one wrong character will keep the preprocessor from translating your pseudo-function. Another drawback comes when you need to do more than a simple translation, e.g., convert a parameter to a character string or code block. This is where the #command, #translate, #xcommand, and #xtranslate directives come into play. These directives allow us to create our own user-defined commands.

The first place you should turn for excellent examples of these four directives is the STD.CH file, which contains dozens of user-defined commands. Actually, as you look closely at STD.CH, you will realize that there are no longer "commands" per se. Every single command gets preprocessed into one or more function calls. This in itself is quite an eye-opening experience.

STD.CH is provided for review purposes only. Its contents are embedded in the Clipper compiler (CLIPPER.EXE) for performance purposes. If you wish to alter any of the standard Clipper command set, we highly recommend that you make a copy of STD.CH. Then be sure to specify this modified STD.CH with the compiler /U option (see Chapter 1).

If you plan to modify only several of the commands, you may wish to put those commands in a header file and #include that in your source code. This principle will be demonstrated in later chapters.

#xcommand and #xtranslate

These directives were added with Clipper 5.01. They are identical to #command and #translate, respectively, with the exception that they require **exact** matches. #command and #translate only require a match on the first four letters of the input text.

These directives are extremely useful in situations such as the following: Suppose that you want to have a preprocessor function called DateWord(), which would return the verbose system date.

```
#translate DateWord() => ;
       cmonth(date()) + ' ' + ltrim(str(day(date()))) + ;
       ', ' + str(year(date()), 4)
function main
? dateword()
return nil
```

Unfortunately, the preprocessor will look at only the first four characters, and thus mistake your DateWord() call for DATE(). This will obviously cause you problems. In fact, this particular example will not even compile because DATE() is used as part of the output text. Therefore, a circular reference will result, closely followed by a stack overflow crash.

We highly recommend that you use #xcommand and #xtranslate rather than #command and #translate. The #command and #translate directives are useful only for compatibility with the dBASE convention of abbreviating commands. But as you have already seen, they can lead to circularity errors and general confusion, so they should be avoided at all costs.

Structure of a user-defined command

The basic syntax of a user-defined command is:

```
#xcommand <input text> => <result text>
#xtranslate <input text> => <result text>
```

A user-defined command consists of three basic parts: the input text, arrow separator ("=>"), and result text.

An important distinction between the #xcommand and #xtranslate directives is that user-defined commands specified with the #xtranslate or #translate directive may appear anywhere in a statement (as can #defines). By contrast, user-defined commands specified with #xcommand or #command MUST be the first non-whitespace characters on a line.

For example, look at the Clipper redefinition of the CLEAR command:

```
#command CLEAR => CLEAR SCREEN ; CLEAR GETS;
```

If you changed this directive to a #translate, and then attempted the following command:

```
@ 10, 12 CLEAR
```

the preprocessor would run into problems. Its first pass would produce the following:

```
@ 10, 12 CLEAR SCREEN ; CLEAR GETS ;
```

CLEAR SCREEN and CLEAR GETS are both #command directives, and therefore cannot appear within a statement in this fashion. However, a bigger problem is that the preprocessor will look at the "CLEAR" in CLEAR SCREEN and attempt to translate that **yet again** based on the #translate CLEAR directive! This will quickly lead to a circularity error that stops compilation dead in its tracks.

Here's a simple rule of thumb for you to use when deciding upon #xcommand or #xtranslate: If the translated expression returns a value, use #xtranslate. Otherwise, use #xcommand.

Input text

This is what the preprocessor will be looking for as it scans your source code. Input text can contain any or all of the following three items:

- **Literal values** — characters that must appear exactly in your input text for the preprocessor to be able to translate it. An example of a literal value is the "@" in the @..CLEAR command:

```
#command  @ <top>, <left>  CLEAR   => ;
__AtClear( <top>, <left>, MaxRow(), MaxCol())
```

- **Words** — keywords that are compared according to the time-honored dBASE tradition (case-insensitive and first four letters only). Therefore, if you write:

```
@ 0, 0 clea
```

  the preprocessor will still be able to translate it according to the #command directive given for @..CLEAR.

- **Match-Markers** — the "parameters" that vary according to the user-defined command. These are treated differently than in #define statements. In a #define, you simply specify a parameter between parentheses:

```
#define TIMES(a, b)    (a) * (b)
```

  When you use a #xtranslate or #xcommand directive, however, you must surround such parameters by "<" and ">" on both sides of the arrow:

```
#xtranslate TIMES( <a>, <b> )  =>  ( <a> ) * ( <b> )
```

Match-markers assign each parameter a name, which you can then refer to in the output (or "result") text. In the TIMES() example shown above, the match-markers mark and assign two chunks of text: **a** and **b.**

Match-markers correspond to **result-markers**, which write the text resulting from the preprocessor's translation. You can easily see from the TIMES() example how **a** and **b** are configured to appear in the resulting preprocessor output for this command. (We'll get to result-markers in more detail in the discussion of **result text**.)

Before we hurl ourselves into the match-marker discussion, let's introduce several important new terms:

- **"stringify"** — convert into a character string
- **"blockify"** — convert into a code block
- **"logify"** — convert into a logical value

We also must ask you to buckle up and put on your thinking caps, because you are about to enter a zone congested with new terminology.

Match-markers

Of the many new Clipper 5 concepts, match-markers (and result-markers) are perhaps the most difficult to grasp. There are many different types of match-markers, all of which serve particular purposes. If you do not fully understand the significance of most of these match-markers, do not spend any time worrying about it. The one that you will rely upon most often is the regular match-marker. As you become more confident with the preprocessor and more sophisticated with your user-defined commands, then you may wish to refer back to this section to learn exactly how and where to use the more specialized match-markers.

Table 7.2 Match-markers

Syntax	*Type*
`<name>`	Regular Match-marker
`<name, ...>`	List Match-marker
`<name: word list>`	Restricted Match-marker
`<*name*>`	Wild Match-marker
`<(name)>`	Extended Expression Match-marker

- The regular match-marker is the most common of the match-markers. It simply matches the next legal expression in the input text. The regular match-marker is used most often with the regular result-marker, but can also be used with the stringify result-markers, and the blockify result-marker. An example of this type of match-marker is the DO WHILE command.

```
#command DO WHILE <exp>  => while <exp>
```

- The list match-marker matches a comma-separated list of expressions. If no input text matches the match-marker, the specified marker name will contain nothing and will thus not be used in the result text. An example of the list match-marker is the ? command, which accepts a list of parameters.

```
#command ? [list, ...>]  =>  QOUT (<list>)
```

- The restricted match-marker is for processing input text that must match one of the words in a comma-separated list. If the input text is not contained in the specified list, the match will fail and the marker name contains nothing. This type of match-marker is most often used with the logify result-marker to write a logical value into the result text. Examples of this match-marker can be found in many of the SET commands, which generally accept the words "ON", "OFF", or a variable name preceded by the macro operator. If a variable is used, it must have a character value, preferably "ON" or "OFF" as you will see from the following example.

```
#command SET CENTURY <x:ON,OFF,&> =>;
__SetCentury(ONOFF( <(x)> ))
y := "on"
set century to &y   // __SetCentury((Upper(y) == "ON"))
```

- The wild match-marker will match any input text from the current position to the end of a statement. This can be used to match input text that may not be a legal expression. A notable example of this usage is the tongue-in-cheek "compatibility" section of STD.CH, which dispenses with numerous dBASE III PLUS commands that are useless to a compiler:

```
#command SET ECHO <*x*>         =>
#command SET HEADING <*x*>      =>
#command SET MENU <*x*>         =>
#command SET STATUS <*x*>       =>
#command SET STEP <*x*>         =>
#command SET SAFETY <*x*>       =>
#command SET TALK <*x*>         =>
```

Therefore, if you included this flippant line in your source code:

```
set echo, is there an echo in here?
```

the preprocessor would output a blank line without even flinching.

The wild match-marker is also used to gather the input text to the end of the statement and write it to the result text using one of the stringify result-markers. An example of this usage is the SET PATH TO command.

```
#command SET PATH TO <*path*>  => Set(_SET_PATH, <(path)> )
```

- The extended expression match-marker will match a regular or extended expression, including filename or path specifications. This allows you to pass a specification without quotes, or in parentheses as an extended expression. You can then use the smart stringify result-marker to ensure that extended expressions will not get stringified. The SET DEFAULT command provides an example of the extended expression match-marker.

```
#xcommand SET DEFAULT TO <(path)> =>;
        set(_SET_DEFAULT, <(path)> )
SET DEFAULT TO c:\app     // set(_SET_DEFAULT, "c:\app")
SET DEFAULT TO ("c:\app") // Set(_SET_DEFAULT, ("c:\app"))
```

Optional clauses

You may specify optional match-clauses by enclosing them in brackets ("[" and "]"). Optional clauses may contain literal values, words, match-markers, and even other optional clauses. There are two types of optional match-clauses:

- A keyword followed by match-marker(s), such as @..GET:

```
#command @ <row>, <col> GET <var> [PICTURE <pic>];
            [RANGE <lo>, <hi>]
```

- A keyword by itself, such as SET KEY TO:

```
#command  set key <n> [TO]  =>  SetKey( <n>, nil)
```

Result text

Result text is the preprocessor's output after translating your source code. It can contain any or all of the following three items:

- **Literal values** — characters that are written directly to the result text. There are examples of literal values in nearly every user-defined command.

- **Words** — keywords and identifiers that are written directly to the output text. Again, there are examples of these in nearly every user-defined command.

- **Result-Markers** — references to match-marker names. As mentioned previously, input text which is matched by a match-marker is written to the result text via a result-marker. As with match-markers, result-markers must be surrounded by "<" and ">".

Result-markers

As with match-markers, there are numerous result-markers. Once again, do not fret if these do not all make sense right now. The one that will be used most often is the regular result-marker. As you write more complex user-defined commands, you can always refer back to this section to learn exactly how and where to use the more specialized result-markers.

Table 7.3 Result-Markers

Syntax	*Type*
`<name>`	Regular result-marker
`#<name>`	Dumb stringify result-marker
`<"name">`	Normal stringify result-marker
`<(name)>`	Smart stringify result-marker
`<{name}>`	Blockify result-marker
`<.name.>`	Logify result-marker

- The regular result-marker writes the matched input text to the result text. It writes nothing if no input text is matched. As with the regular match-marker, this is the most common result-marker and therefore most likely to be used. This result-marker is used most often with the regular match-marker but can be used in combination with any of them. We will rewrite our TIMES() pseudo-function to give an example of regular result-markers.

```
#xtranslate TIMES( <a> , <b> )  =>  ( <a> * <b> )
```

- The dumb stringify result-marker turns the matched input text into a character string and writes it to the result text. If no input text is matched, this result-marker will write a null string ("") to the result text. If used in conjunction with the list match-marker, the list will be converted to a character string and written to the result text. An example of this result-marker is the SET COLOR TO command, which accepts an unquoted color specification.

```
#command SET COLOR TO [<*spec*>]   =>  SetColor(#<spec> )
set color to w/b                   //  SetColor("w/b")
```

Note: the dumb stringify result-marker is not quite as dumb as you might think. It is intelligent enough to detect the presence of string delimiters in your character string and work around that.

```
#xtranslate show_msg( <message> ) => showmsg(#<message> )
show_msg(test1)                   // showmsg("test1")
show_msg("test2")                 // showmsg('"test2"')
```

- The normal stringify result-marker is very similar to the dumb stringify result-marker. The differences are that if no input text is matched, this result-marker writes nothing (rather than a null string). Also, if used with the list match-marker, each element in the list will be stringified, as opposed to the entire list as one entity. The RELEASE command provides an example of the normal stringify result-marker.

```
#command RELEASE <vars,...> =>  __MXRelease( <"vars"> )
release mvar                //  __MXRelease("mvar")
release mvar, mvar2, mvar3  //  __MXRelease("mvar",;
                            //              "mvar2","mvar3")
```

- The smart stringify result-marker converts the matched input text to a character string only if it is enclosed in parentheses. If no input text is matched, nothing is written to the result text. If used in conjunction with the list match-marker, each element in the list is stringified using this same rule and written to the result text.

The smart stringify result-marker is designed specifically to support extended expressions for commands other than SETs. One such use is the ERASE command.

```
#command  ERASE <(file)>    =>  FErase( <(file)> )
mvar := "temp.dbf"
ERASE temp.dbf              //  FErase("temp.dbf")
ERASE (mvar)                //  FErase((mvar))
```

- The blockify result-marker converts the matched input text to a code block. If no input text is matched, nothing is written to the result text. If used with the list match-marker, each element in the list is converted that way. Many Clipper 5 functions rely upon code blocks, which makes this result-marker a lot more important than you might imagine. An example of blockifying is the SET FILTER command.

```
#command SET FILTER TO <xpr> => dbSetFilter(<{xpr}>,<"xpr"> )
set filter to ! deleted //  dbSetFilter({|| ! deleted()},;;
                        //              "! deleted()")
```

- The logify result-market writes true (.T.) to the result text if input text is matched, or false (.F.) if it is not. The input text itself is not written to the result text. As mentioned above, this type of result-marker is best used with the restricted match-marker. An example demonstrating both of these is the SET MESSAGE TO command.

```
#command SET MESSAGE TO <n> [<cent: CENTER, CENTRE>] => ;
set(_SET_MESSAGE, <n> ) ; Set(_SET_MCENTER, <.cent.> )
set message to 24         // Set(_SET_MESSAGE, 24) ;;
                          // Set(_SET_MCENTER, .F.)
set message to 24 center  // Set(_SET_MESSAGE, 24) ;;
                          // Set(_SET_MCENTER, .T.)
```

As you have seen in these examples, the output text can contain more than one statement. Each statement must be separated by a semicolon. When you start writing more complex user-defined commands that need continuation lines, be sure not to skimp on the semicolons.

```
#xtranslate endsearch() => mrow := row() ; mcol := col() ; ;
                          searching := .f. ; searchstr := '' ; ;
                          devpos(ntop, (maxcol() + 1) / 2 - 11) ; ;
                          devout(replicate(chr(205), 22)) ; ;
                          devpos(mrow, mcol)
```

If you want to use a less than symbol ("<") or brackets ("[", "]") in the output text, you must preface them with a backward slash ("\"). The backslash is necessary because those characters all carry special meaning to the preprocessor. "<" indicates the beginning of a match-marker, and brackets indicate optional clauses.

Please feel free to use white space liberally in your #xcommand and #xtranslate constructs. The preprocessor needs it to be able to properly convert everything. In fact, the only place where you would not want white space is between the angle brackets and the match markers (i.e., use "<msg>" rather than "< msg >"). It will compile properly, but will detract from the readability of your code.

One of the benefits to writing your own user-defined commands is to minimize naming conflicts between your functions and others of the same name. Center() is a great example of this — nearly every Clipper programmer has a Center() function, and they all use slightly different syntax. This can create massive problems when you call your Center() function but link in someone else's. You will get run-time type mismatch errors and not understand why.

However, if you make Center() into a user-defined command as we did earlier, *it no longer exists as a function.* All centering is done in-line, which therefore completely eliminates any potential naming conflicts. You will never have to worry again about accidentally invoking someone else's Center() function.

Scope of user-defined commands

When you write a user-defined command with the #xcommand or #xtranslate directives, it will remain visible from that line until the end of that PRG file. It will not be visible in other PRG files. The following code fragment demonstrates this principle.

Listing 7.2 Scope of user-defined commands

```
/* MAIN.PRG */
#xcommand REDRAW => @ 0, 0, maxrow(), maxcol();
          box replicate(chr(176),9)

function main
redraw
do test
return nil
* eof: main.prg

/* TEST.PRG */
redraw
return nil
* eof: test.prg
```

The preprocessor will be unable to translate the REDRAW command in TEST.PRG, which will lead to a compile error ("statement unterminated").

The only exception to this rule of visibility are user-defined commands in STD.CH (or an alternate standard rules file that you specify with the /U compile option).

Precedence

Multiple directives per statement

The preprocessor translates the three primary directives in this order: #define; #translate; #command. As each directive is encountered, the preprocessor will translate it appropriately and then rescan that line of code for any other directives. Look at the following directives:

```
#define OFFSET   20
#define FROMBACK(a) len(a - OFFSET)
#xtranslate addem( <a> ) => aeval( <a>, { | ele | msum += ele },;
           FROMBACK( <a> ))
AddEm(myarray)
```

When the preprocessor encounters the AddEm() user-defined command, it will convert it to:

```
aeval(myarray, { | ele | msum += ele }, FROMBACK( myarray))
```

It will then make another pass at that line, which will reveal FROMBACK. Because this has been #defined as a pseudo-function, it too will be translated:

```
aeval(myarray, { | ele | msum += ele }, len(myarray - OFFSET))
```

Finally, the preprocessor will act upon the #definition of OFFSET to translate that as well:

```
aeval(myarray, { | ele | msum += ele }, len(myarray - 20))
```

Since there are no more directives to be processed for this line, the preprocessor considers this a job well done and moves on.

Most recent definition

The preprocessor will consider the most recent definition of each directive when translating your code. This means that, for example, if you #define a manifest constant in your source code and then #include a header file that redefines it, the redefinition will be used. The following example demonstrates this:

```
// TEMP.PRG
#define ELEMENTS 5
#xtranslate Center( <a> ) => ;
            space(int((80 - len( <a> ))/2))
#translate READ => readmodal(getlist) ; ;
                   aadd(mastergets, getlist)
#include "MYSTUFF.CH"

function main
memvar getlist
local a[ELEMENTS], mastergets := {}, string := space(40)
cls
@ 2, 20 get string
read
string = trim(string)
@ 3, center(string) say string
return nil
* end of file TEMP.PRG

// MYSTUFF.CH
#define ELEMENTS 100
#xtranslate Center( <row>, <msg> ) => ;
@ <row>, space(int((80 - len( <msg> )) / 2) say <msg>
```

The preprocessed output will look like this:

```
#line 1 "d:\MYSTUFF.CH"
#line 6 "d:\TEMP.PRG"
function main
memvar getlist
local a[100], mastergets := {}, string := space(40)
Scroll (); SetPos(0, 0)
SetPos(2, 20) ; AAdd(GetList, _GET_(string, "string",,,))
readmodal(getlist) ; aadd(mastergets, getlist)
string = trim(string)
DevPos(3, space(int((80 - len( string ))/2))) ; ;
DevOut(string)
return nil
```

Although the manifest constant ELEMENTS was defined in TEMP.PRG, the preprocessor used the value found in MYSTUFF.CH since that file was #included after the #define statement. In similar fashion, the user-defined command Center() gets pulled in from MYSTUFF.CH and thus overrides the original #xtranslate directive in TEMP.PRG.

Note that unlike #defines, if you redefine a #command or #translate directive you will **not** get a compiler warning.

Did you notice the redefinition of the Clipper READ command? You can override standard Clipper commands in this manner. Clipper's standard rule set (as seen in STD.CH) is loaded at the beginning of the compilation process. In accordance with the rule of most recent definition, the preprocessor will therefore use your redefinition of the READ command.

The #error directive

This directive was added with the release of Clipper 5.01. If the #error directive is encountered by the compiler, it will cause the compilation process to stop dead in its tracks.

Why might you want to do this? The best reason is that #error allows you to bullet-proof your code in situations where you absolutely, positively must depend upon certain manifest constants. There are at least two instances where this would be important:

1. If you are working with other programmers in a team environment.

2. If your code assumes that certain directives will be passed on the command-line with the /D compiler switch.

The following code fragment shows how you can ensure that the manifest constant ITERATIONS exists:

```
#ifndef ITERATIONS
  #error Missing ITERATIONS - try again, bucko
#endif
```

This directive is used extensively in the RESERVED.CH header file, which is used to preclude naming collisions between your functions and reserved Clipper function names. Please refer directly to the RESERVED.CH file for numerous examples.

Examples

Writing bilingual programs

Suppose that you have a vertical market application that will be used by both English-speaking people and Francophones. There are three basic ways of addressing this problem:

1. Maintain two completely separate versions of your source code. This is obviously not very practical and is hardly worth doing.

2. Strip all static (user interface) text out of the program and create text or MEM files for each language. Restructure the program to read in the variables from this file at the beginning and make use of it throughout.

   This is infinitely preferable to method #1. However, it has several potential drawbacks, which include the performance penalty you must incur for the disk access, and potential simultaneous file access problems on networks. But for a single-user system running on a relatively quick CPU (for example, a 386/25 Mhz or above), this is a very adequate solution.

3. Use the preprocessor!

You can probably guess that we plan to discuss the third option, since that is the whole point of this dissertation. It's really quite easy — we rely upon the #ifdef directive, whose power we only briefly hinted at earlier. For example:

Listing 7.3 Bilingual manifest constants

```
#ifdef FRENCH
   #define M_NETERR    "Dossier entrain d'être utiliser, " +;
                       "ne pouvez pas modifier"
   #define M_CONTINUE  "Voulez-vous continuer? (O/N)"
   #define M_TOF       "Commencement de fichier!"
   #define M_BOF       "Fin de fichier!"
   #define M_PRINT     "Imprimante ou fichier?"
   #define M_CONFIRM   "Etes-vous sur?"
   #define M_NOTFOUND  "Pas Trouver!"
   #define M_ADDING    "Ajoute dossier - ^W pour sauver; "+;
                       "Esc pour sortir"
   #define M_EDITING   "Modifie dossier - ^W pour sauver; "+;
                       "Esc pour sortir"
#else
   #define M_NETERR    "Could not lock record at this time - "+;
                       "edits not saved"
   #define M_CONTINUE  "Would you like to continue? (Y/N)"
   #define M_TOF       "Top of file!"
   #define M_BOF       "Bottom of file!"
   #define M_PRINT     "Print to printer or file?"
   #define M_CONFIRM   "Are you sure?"
   #define M_NOTFOUND  "Not Found!"
   #define M_ADDING    "Add record - ^W to save; Esc to exit"
   #define M_EDITING   "Edit record- ^W to save; Esc to exit"
#endif
```

Next, you refer to these messages by the manifest constants that identify them. For example, here is the code for a failed SEEK:

```
seek whatever
if ! found()
   Center(maxrow(),M_NOTFOUND)
endif
```

Now you can switch effortlessly between the two languages by using the /D compiler option. If you want to use French, compile with the following command:

```
clipper myprog /dFRENCH
```

The preprocessor will detect the presence of the FRENCH identifier, and use only the French manifest constants.

If you want to use English, just compile without using the /D option. Because the FRENCH identifier will be undefined, the preprocessor will use the English manifest constants.

Box drawing

Are you as tired of drawing box outlines as we are? Here are some user-defined commands and a handy literal array to address this.

Before proceeding, remember that Nantucket does include a BOX.CH header file that has some useful manifest constants representing box outlines. Our main complaint with their definitions is that none of them include the ninth character (which fills the box), so you therefore have to add it yourself for the interior of your boxes to be cleared. This leads directly into our first two examples:

```
#xtranslate SINGLEBOX( <top>, <left>, <bottom>, <right> ) => ;
            @ <top>, <left>, <bottom>, <right> BOX "┌─┐│┘─└│ "
#xtranslate DOUBLEBOX( <top>, <left>, <bottom>, <right> ) => ;
            @ <top>, <left>, <bottom>, <right> BOX "╔═╗║╝═╚║ "
```

So if you want to draw a single box, all you need to do is:

```
singlebox(5, 0, 10, 50)
```

It does not get much easier than that! But if you want more flexibility with the box outline, here is another example that you can use:

```
#define BOXFRAMES{ '╔═╗║╝═╚║ ', '┌─┐│┘─└│ ', '╒═╕│╛═╘│ ',;
               '╓─╖║╜─╙║ ', '█▀██▀▀▄█ ', SPACE (9) }
```

This is not an in-line function; instead, it is a literal array containing six elements. Whenever you refer to BOXFRAMES in your source code, the preprocessor will substitute this array in its place.

As you can tell, each of these six elements is a different sort of box outline. How would you use this? Let's suppose that you wanted to draw a box with a thick border. The thick border is the fifth element in the array. Here's the call:

```
@ 5, 0, 10, 50 box BOXFRAMES[5]
```

Again, this is quite easy and very clean. To make matters even simpler, you could set up manifest constants referring to each type of box outline so that you would not even have to remember the array element numbers:

```
#define B_DOUBLE         1
#define B_SINGLE         2
#define B_DOUBLESINGLE   3
#define B_SINGLEDOUBLE   4
#define B_THICK          5
#define B_NONE           6
@ 5, 0, 10, 50 box BOXFRAMES[B_THICK]
```

Date literals

This idea comes courtesy of Don Caton. Mr. Caton is a very clever fellow who actually found something to like about dBASE IV: its literal representation of dates. It is old news that to declare a date variable in Clipper (and dBASE III PLUS), you had to rely upon the CTOD() function.

```
mdate := ctod("03/01/91")
```

dBASE IV lets you declare that date like so:

```
mdate = {03/01/91}
```

Don seized the opportunity to use the #xtranslate directive and the dumb stringify result-marker to provide this functionality for Clipper 5 programmers.

Original (.PRG):

```
#xtranslate { <m> / <d> / <y> } =>;
          ctod(#<m> + "/" + #<d> + "/" + #<y> )
mdate := {03/01/91}
```

Preprocessed output (.PPO):

```
mdate := ctod("03" + "/" + "01" + "/" + "91")
```

As you can see, the month, day, and year do not need to be enclosed in quotes because the dumb stringify result-marker will do that for you. Yes, this still calls the CTOD() function, but it cleans up your code and saves you time coding. (Note that there is an infinitesimal chance of the preprocessor getting confused between a date literal and, say, a literal array, but it has not happened to any of us.)

DEFAULT

In prior versions of Clipper, the testing of parameters passed to functions was accomplished primarily with the Clipper PCOUNT() and TYPE() functions. Although we have learned to live with this arrangement, Clipper 5 renders it obsolete with the NIL data type.

In Clipper 5 you skip unnecessary parameters simply by placing a comma in the parameter list. You no longer need to use a null string as in Summer '87. If a parameter in the formal list has been omitted, it will merely be initialized within the function with the value of NIL.

This sample function, which displays a boxed message on the screen, accepts three parameters. Let's assume that the first parameter (the message) will always be passed. The second and third parameters are completely optional, and affect the color of the box and message, respectively. If these two parameters are not passed, they will be assigned the value of NIL within ShowMsg(). It is therefore very easy to test for whether the parameters were passed or not.

Listing 7.4 ShowMsg()

```
function ShowMsg(msg, boxcolor, msgcolor)
local buffer, leftcol := int(76 - len(msg)) / 2
// assign box color if not passed as parameter
if boxcolor == nil
   boxcolor := 'w/r'
endif
// assign message color if not passed as parameter
if msgcolor == nil
   msgcolor := '+w/r'
endif
buffer := savescreen(11, leftcol, 13, 80 - leftcol)
@ 11, leftcol, 13, 80 - leftcol box "╔═╗║╝═╚║ " color boxcolor
@ 12, leftcol + 2 say msg color msgcolor
inkey(2)
restscreen(11, leftcol, 13, 80 - leftcol, buffer)
return nil
```

This can be made even more succinct with the preprocessor. Rather than having to code the IF..ENDIF logic for each parameter, we can simply define a user-defined command to do the dirty work for us:

```
#xcommand DEFAULT <p> TO <v> [, <p2> TO <v2> ] => ;
     <p> := if( <p> == NIL, <v>, <p> );
     [; <p2> := if( <p2> == NIL, <v2>, <p2> ) ]
```

This tests the parameter (<p>) against NIL, and assigns it the default value (<v>) if that's what it is. Otherwise, the parameter's value is left unchanged.

The use of the brackets in the result text indicates that the optional clause may be repeated. You can therefore chain values together in the same statement, as we are about to see.

Now, instead of:

```
if boxcolor == NIL
   boxcolor := 'w/r'
endif
if msgcolor == NIL
   msgcolor := '+w/r'
endif
```

you can simply write:

```
default boxcolor to 'w/r', msgcolor to '+w/r'
```

which eliminates four lines of code without sacrificing one iota of readability.

Free-format function parameter lists

But wait, that's not all you can do with NIL. You can also construct free-format function parameter lists. In this example, you could write a preprocessor translation directive for ShowMsg() that would allow you to pass the three parameters in any order you wanted. The only provision would be that you would have to identify the parameter.

```
#xtranslate showmsg([MSG <msg>] [BOXCOL <boxcol>];
                     [MSGCOL <msgcol>]) ;
            => showmsg( <msg>, <boxcol>, <msgcol> )
```

Each clause should be surrounded by brackets to indicate that it is optional, and there is no need to separate them with commas.

The following statements:

```
showmsg(MSGCOL 'w/b' MSG 'this is the message' BOXCOL 'w/r')
showmsg(MSGCOL 'w/b' BOXCOL 'w/r' MSG 'this is the message')
showmsg(BOXCOL 'w/r' MSGCOL 'w/b' MSG 'this is the message')
```

will all be neatly translated into:

```
showmsg("this is the message", "w/r", "w/b")
```

Omitting a parameter is no big deal, since Clipper 5 lets us skip parameters.

Original (.PRG)

```
showmsg(MSGCOL 'w/b' MSG 'this is the message')
```

Preprocessed output (.PPO)

```
showmsg('this is the message',, 'w/b')
```

With this syntax the second parameter (box color) gets passed as a NIL. Because we have already configured our logic to test each parameter against NIL, there is no possible way that anything can go wrong.

Adding extensions to filenames

Most Clipper programs prompt the user to enter a filename. They are often allowed to enter an optional extension, but if they do not, the program must add one itself. The following user-defined command fills the bill quite neatly:

Original: (.PRG)

```
#xtranslate AddExtension( <file>, <ext> ) => ;
      <file> := upper( <file> ) + if(! "." + upper( <ext> ) $ ;
                upper( <file> ), "." + upper( <ext> ), '')

AddExtension(mfile, 'dbf')
```

Preprocessed: (.PPO)

```
mfile := upper(mfile) + if(! "." + upper("dbf") $ upper(mfile), ;
                "." + upper("dbf"), "")
```

No more STRPAD()

Nantucket should have included this directive in STD.CH. If you used the STRPAD() function in Clipper Summer '87, you may have already discovered that it does not exist in Clipper 5. However, Clipper 5 does provide the PADR() function, which does everything that STRPAD() did. Rather than having to go through your code and change each occurrence of STRPAD() to PADR(), you could write a simple translation to do it for you:

```
#xtranslate strpad( <msg>, <length> ) => padr( <msg>, <length> )
```

Alias expressions

As you may recall from Chapter 5, the alias operator ("->") allows you to refer to a field or evaluate an expression in an unselected work area. The alias operator will automatically SELECT the desired work area, perform the operation, and reselect the previous work area. This allows you to compact your code by requiring fewer explicit SELECT statements. (The ALIAS clause exists only with the SKIP command.)

These are some examples to get you started. The basic idea is to add the optional clause "[ALIAS <a>]" to the input text, and the corresponding optional clause "[<a> ->]" to the result text. You also must make sure to surround the relevant function call with parentheses.

Listing 7.5 Alias expressions

```
#command SEEK <xpr>   [ALIAS <a>] =>  [<a> ->] (dbSeek( <xpr> ))
#command GOTO <n>     [ALIAS <a>] =>  [<a> ->] (dbGoto( <n> ))
#command GO <n>       [ALIAS <a>] =>  [<a> ->] (dbGoto( <n> ))
#command GOTO TOP     [ALIAS <a>] =>  [<a> ->] (dbGoTop())
#command GO TOP       [ALIAS <a>] =>  [<a> ->] (dbGoTop())
#command GOTO BOTTOM  [ALIAS <a>] =>  [<a> ->] (dbGoBottom())
#command GO BOTTOM    [ALIAS <a>] =>  [<a> ->] (dbGoBottom())
#command CONTINUE     [ALIAS <a>] =>  [<a> ->] (dbContinue())
#command APPEND BLANK [ALIAS <a>] =>  [<a> ->] (dbAppend())
#command UNLOCK       [ALIAS <a>] =>  [<a> ->] (dbUnlock())
#command PACK         [ALIAS <a>] =>  [<a> ->] (__dbPack())
#command ZAP          [ALIAS <a>] =>  [<a> ->] (__dbZap())
#command DELETE       [ALIAS <a>] =>  [<a> ->] (dbDelete())
#command RECALL       [ALIAS <a>] =>  [<a> ->] (dbRecall())

function Main
use invoices new
set index to invoices
use customer new
seek customer->Custno alias invoices
delete alias invoices
go top alias invoices
return nil
```

As mentioned earlier, you should not modify the STD.CH file directly. If you want to use the alias clause as outlined here, make a copy of STD.CH, rename it to something else (ALIAS.CH might be appropriate), and modify the database commands accordingly. Then strip out everything except what you have changed. By #including this new header file in your source code, your database commands will override the Clipper defaults.

Throughout this book we advise you against modifying the STD.CH file. Here are the reasons:

- STD.CH is provided for reference purposes only. The rules therein are actually embedded within the compiler (CLIPPER. EXE) to enhance performance.

- If you made changes to STD.CH, you would then have to use the compiler /U option to use your modified file as the standard rules file. This will cause your compilation process to be significantly slower.

- Rather than make changes to STD.CH and use it with /U as outlined above, we recommend that if you plan to modify one or two commands, you should place those modifications in your own header file and #include that in your source code. Because the standard rules are loaded first, anything that you redefine will override the standard definition. This will allow you to use your modified commands, without sacrificing speed during compilation.

Grabbing text/color attributes

If you need to check the text and/or color attribute at a particular location on the screen, you can use SAVESCREEN() to save that element and then parse it. However, rather than clutter your code with lots of messy SAVESCREEN() calls, use these two user-defined commands, TextAt() and ColorAt().

Original: (.PRG)

```
#xtranslate TextAt( <r>, <c> ) => ;
                    substr(savescreen( <r>, <c>, <r>, <c> ), 1, 1)
#xtranslate ColorAt( <r>, <c> ) => ;
                    substr(savescreen( <r>, <c>, <r>, <c> ), 2, 1)
x := TextAt(10, 0)
```

Preprocessed: (.PPO)

```
x := substr(savescreen(10, 0, 10, 0), 1, 1)
```

Music Maestro please

Even if you scoff at the prospect of using audio feedback in your programs, it still behooves you to look at the coding techniques presented here. We have included three familiar musical themes: "Charge", "NannyBoo", and "TheFifth" (opening notes to Beethoven's Fifth). When you call one of these commands, the preprocessor will convert it to a multi-dimensional array (containing note frequency and duration), which in turn gets passed to the function Tunes(). Tunes() is not actually a function, so the preprocessor translates it into an AEVAL() statement which invokes the TONE() function to play one note for each element in the array.

Because the #xcommand directive is used, these song names must begin a statement rather than be contained within it.

Listing 7.6 Preprocessed music

```
#xcommand Charge   => tunes({ {523,2}, {698,2}, {880,2}, ;
                         {1046,4}, {880,2}, {1046,8} } )
#xcommand NannyBoo => tunes({ {196,4}, {196,4}, {164,4}, {220,4}, ;
                         {196,8}, {164,8} } )
#xcommand TheFifth => tunes({ {392,2}, {392,2}, {392,2}, {311,10},;
                         {15,12}, {349,2}, {349,2}, {349,2},;
                         {293,10} } )
#xtranslate tunes( <a> ) => aeval( <a>,;
                          { | a | tone(a\[1], a\[2]) } )
```

```
function Music
charge
inkey(0)
nannyboo
inkey(0)
thefifth
inkey(0)
return nil
```

This should whet your appetite for AEVAL() and multi-dimensional arrays, which are covered at length in the next two chapters, respectively.

Summary

You are now ready to put the Clipper 5 preprocessor to work for you. You understand how to define manifest constants both in your source code and on the command-line. You know how to conditionally compile your source code via manifest constants.

A thorough knowledge of the preprocessor is a prerequisite for many of the topics that await you. For example, we will use the preprocessor to write a replacement for Clipper's light-bar menu system (@..PROMPT and MENU TO) in Chapter 14. We will also rely upon the preprocessor to add more clauses to the @..GET command in Chapter 26.

CHAPTER 8

Code Blocks

Code blocks are a new data type in Clipper 5. They often engender a great deal of fear and confusion at first, because they are such a radically different concept.

Therefore, this chapter will shed some much-needed light on these maligned and misunderstood creatures. When you have finished, you should be ready and willing to make the code block your ally instead of your enemy.

We will also discuss in depth the numerous Clipper functions that use code blocks, including EVAL(), AEVAL(), DBEVAL(), FIELDBLOCK(), FIELDWBLOCK(), and SETKEY().

Overview

Code blocks are an integral (and inescapable) part of Clipper 5. Even if you never explicitly write a code block, you can bet that the preprocessor will be turning many of your commands into code blocks, so you might as well learn how to use them to your advantage.

Code blocks are a data type that contain compiled Clipper code. They can be compiled either at compile-time with the rest of your Clipper code, or at run-time with the use of the "&" operator.

This is a code block in its rawest form:

```
{ | [<argument list>] | <expression list> }
```

Code blocks look quite similar to Clipper 5 literal arrays. Both code blocks and arrays begin with an open curly brace ("{") and end with a closed curly brace ("}"). But code blocks differentiate themselves by including two "pipe" characters ("|") directly after the opening brace. These pipe characters delimit an optional <argument list>, which would then be passed to the code block upon evaluation. The <argument list> must be comma-delimited (e.g., "a, b, c...").

Although the white space between the pipe characters and braces is purely optional, we highly recommend you use it for the sake of readability.

The <expression list> is, obviously enough, a comma-delimited list of any valid Clipper expressions. These can run the gamut, as you will quickly discover.

Writing a code block

There are three methods for writing a code block:

- To be compiled into a code block at compile-time, for example:

  ```
  local myblock := { | | fname }
  ```

- As a character string, which can be compiled to a code block at run-time. For such compilation you can use the & operator. Yes, the same one that is used for macros. But remember that this is not the same thing as macro substitution! Suppose you wanted to set up a TBrowse() object to browse a database. You would need to establish a column for each field in the database. When setting up TBrowse() columns, you must specify a code block, which when evaluated contains the contents of that column. If you knew in advance that our database contained the fields FNAME, LNAME, and SSN, it would be a simple matter to write the code blocks so that they could be compiled at compile-time:

```
local x, browse := TBrowseDB(3, 19, 15, 60), column
use test new
column := TBColumnNew("FNAME", { | | fname } )
```

```
browse:AddColumn(column)
column := TBColumnNew("LNAME", { | | lname } )
browse:AddColumn(column)
column := TBColumnNew("SSN", { | | ssn } )
browse:AddColumn(column)
do while ! browse:stabilize()
enddo
```

However, let us further suppose that we wish this routine to be generic. We therefore cannot hard-code field names, because the structure will be unknown until run-time. Listing 8.1 shows how we could approach it.

Listing 8.1 Generic TBrowse column setup

```
local x, browse := TBrowseDB(3, 19, 15, 60), column
use test new
for x := 1 to fcount()
   column := TBColumnNew(field(x), &("{ | | " + field(x) + "}"))
   browse:AddColumn(column)
next
do while ! browse:stabilize()
enddo
```

(Note: Later in this chapter we will discuss how to accomplish this same thing with the FIELDWBLOCK() function.)

The Clipper FIELD() function returns the name of the field based at the ordinal position in the database structure. For example, FIELD(2) will return the name of the second field in the database ("LNAME" in our little example).

Caveat: If you compile a code block at run-time that refers to a static or local variable, it will crash unless you pass that variable as an argument. The following code will not run:

```
function Main
static x := 5, y, z
```

```
y := &("{ | | x * 200 }")
? eval(y)   // boom
```

This is because when you attempt to compile the code block with the "&" operator, the variable will not be resolvable because **local** and **static** memory variables do not have entries in the symbol table.

The workaround is to structure your code block to accept arguments, and then pass any static/local variables as parameters at evaluation.

```
function Main
static x := 5, y, z
y := &("{ | x | x * 200 }")
? eval(y, x)
```

Because **x** is declared as an argument to this code block, it will be treated as a variable local to that code block. All you need to do then is make sure to pass an argument when you evaluate the code block.

Note the use of **x** within the code block. This **x** will be local to the code block, and is therefore not related in any way to the variable **x**.

- Don't dirty your head with the mechanics of code blocks and let the preprocessor write them all for you. For example, if you write the following code:

```
index on fname to customer
```

the preprocessor will dedicate a code block in your honor:

```
dbCreateIndex( "customer", "fname", {|| fname},;
               if(.f., .t., nil))
```

Code blocks have much in common with inner city cockroaches: You can neither run nor hide from them. Thankfully, code blocks are a lot more fun and a million times more useful than cockroaches (which is why, if you have read this far, you should keep reading and stop playing with your pet cockroach).

Evaluating code blocks

The only operation that you can perform on a code block is *evaluation*. You can think of evaluation as being analogous to calling a function and returning a value from it. Code blocks can be evaluated by the EVAL(), AEVAL(), or DBEVAL() functions. They are also evaluated internally when you pass them as parameters to functions that can use them.

When evaluated, code blocks return the value of the rightmost expression within them. For example, if you create the following code block:

```
local myblock := { | | mvar }
```

When you evaluate this code block, it will return the current value of **mvar**. If **mvar** is undefined at the time that you evaluate the code block, you will get an undefined error.

```
local myblock := { | | mvar }
local mvar := 500
local x
x := eval(myblock)
? x                          // output: 500
```

Remember that code blocks can contain any valid Clipper expressions. This means that you can get considerably fancier with them than we have dared thus far. For example:

```
local myblock := { | | qout(var1), qqout(var2), 500 }
local var1 := "Clipper ", var2 := "5"
x := eval(myblock)                // "Clipper 5"
? x                               // 500
```

Look again at that last statement. How does **x** get the value of 500? When you evaluate a code block with EVAL(), it returns the value of the last (or rightmost) expression within it. Because the last expression in **myblock** is 500, the variable **x** is assigned that value.

Using code blocks without parameters

These are examples of simple code blocks that do not use parameters. The first simply outputs a value to the console:

```
local myblock := { | | qout(mvar) }, mvar := "testing"
eval(myblock)    // "testing"
```

This returns a constant (5000), which upon evaluation is assigned to the variable X.

```
local myblock := { | | 5000 }
x := eval(myblock)
? x                 // 5000
```

This crashes upon evaluation because the variable **x** has not been defined.

```
local myblock := { | | x++ }
for y := 1 to 100
   eval(myblock) // boom!
next
? x
```

This is the fixed version of the last example. X is defined, and there is joy in Mudville.

```
local myblock := { | | x++ }, x := 1  // much nicer thanks
for y := 1 to 100
   eval(myblock)
next
? x                 // output: 101
```

This is an example of calling one of your own functions from within a code block:

```
local myblock := { | | BlueFunc() }
eval(myblock)   // calls BlueFunc() which displays a message
return nil

static function BlueFunc
? "here we are in a BlueFunc() - will we ever escape?"
inkey(5)
return nil
```

Using code blocks with parameters

Just as with functions, there is far greater power to harness with code blocks when you begin passing parameters. Writing a parameter list for a code block is nearly identical to writing one for a function. However, because it is harder to conceptualize in the linear world of a code block, we'll write a simple code block and then rewrite it as a function:

```
local myblock := { | a, b, c | max(a, max(b, c)) }

function MMax(a, b, c)
return max(a, max(b, c))
```

As you can readily see, the function MMax() returns the highest of the three parameters passed to it. Evaluating the code block **myblock** will return exactly the same thing. However, we must first slip past another stumbling block: how to pass parameters to a code block. It is actually quite simple; the EVAL() function accepts optional parameters after the name of the code block. Each such optional parameter represents a parameter to be passed to the code block. For example, if you write:

```
eval(myblock, 20)
```

you are in effect passing the numeric parameter 20 to the code block defined as **myblock**. Let's have another look at our MMax() function and code block so that you can get a feel for passing parameters with EVAL():

```
local myblock := { | a, b, c | max(a, max(b, c)) }
? MMax(20, 100, 30)              // 100
? eval(myblock, 20, 100, 30)     // 100
```

Do you remember the BlueFunc() that we were just in? (Hope you're feeling better now.) Let's modify the function and the code block to accept a parameter which will dictate how long to wait for a keypress.

```
local myblock := { | x | BlueFunc(x) }
eval(myblock, 20)   // calls BlueFunc() and will wait 20 seconds
return nil

static function BlueFunc(delay)
? "we're in a BlueFunc() for " + ltrim(str(delay)) + " seconds"
inkey(delay)
return nil
```

Here is a code block that accepts up to three parameters and displays them on the screen:

```
local myblock := { | a, b, c | qout(a, b, c) }
eval(myblock, 1, 2, 3)      // 1 2 3
x := eval(myblock, 1, 2)    // 1 2 nil
? x                         // nil
```

You already know why the second EVAL() statement outputs 1, 2, and NIL, right? It is because any declared parameters that are not received are initialized to NIL. Because **myblock** expects three parameters (**a**, **b**, **c**), and we only pass two, **c** gets initialized to NIL. But here's a trick question for you: Do you know why **x** takes the value of NIL? No, it has nothing to do with the fact that we passed too few parameters. Rather, it is because the code block returns the value of the expression QOUT(**a**, **b**, **c**). The Clipper 5 QOUT() function always returns NIL. (If you already knew this, give yourself a pat on the back.)

Important Note: Any arguments that you specify in a code block are automatically given local scope. Such arguments will not be visible to any nested code blocks! This merits another example:

```
local firstblock := { | | qout(x) }
local myblock := { | x | x++, eval(firstblock) }
eval(myblock, 3)
```

This program will crash when you attempt to evaluate **firstblock**. It would seem that the argument **x** in **myblock** should be visible within **firstblock**. But do not be fooled — **x** is local to **myblock** and is therefore NOT visible to **firstblock**.

Functions that crave code blocks

EVAL(<block>, [<arg list>])

You should have already surmised that EVAL() evaluates a code block, which you pass to it as the <block> parameter. The optional parameter <arg list> is a comma-delimited list of parameters to be passed to the code block when you evaluate it.

Return value: As we mentioned previously, EVAL() returns the value of the last (rightmost) expression within the block.

AEVAL(<array>, <block>, [<start>], [<count>])

AEVAL() is similar to EVAL() but is specially designed to work with arrays. It evaluates a code block (specified by the <block> parameter) for each element in the array (specified by the <array> parameter). You may optionally specify a <start> element, and a number of elements (<count>) to process. If you do not use these optional parameters, AEVAL() will begin with the first element in the array and process all of them.

The following AEVAL() is a real workhorse; it determines the maximum, minimum, and sum of all elements in the array MYARRAY:

```
local myarray := { 75, 100, 2, 200, .25, -25, 40, 52 }, ;
      nmax, nmin, nsum := 0
nmax := nmin := myarray[1]
aeval(myarray, { | a | nmax := max(nmax, a),;
                       nmin := min(nmin, a),;
                       nsum += a } )
? "Maximum value:", nmax        // 200
? "Minimum value:", nmin        // -25
? "Total amount: ", nsum        // 444.25
```

AEVAL() automatically passes two parameters to the code block: <value> and <number>. <value> is the value of the array element being processed. <number> is the number of the array element being processed. You have already seen how <value> is used, but why should we bother with <number>? Suppose that you want to increment each element in MYARRAY. You would probably write your code block like this:

```
aeval(myarray, { | a | a++ } )
aeval(myarray, { | a | qout(a) } ) // display results
```

Surprise, surprise! This will not do a single thing to the elements of the array, because they are passed by value (not reference) to the code block. (See Chapter 9 for a thorough discussion of arrays.) Passing by value means that the code block makes a copy of the array element, and any manipulation done within the code block is performed on the copy rather than the genuine article. Let's try it again with the <number> parameter:

```
aeval(myarray, { | a, b | myarray[b]++ } )
aeval(myarray, { | a | qout(a) } )
```

Return value: AEVAL() returns a reference to the array you ask it to process.

DBEVAL(<block>, [<for>], [<while>], [<next>], [<record>], [<rest>])

DBEVAL() is similar to AEVAL(), except that it deals with databases rather than arrays. It also provides far greater control, including FOR, WHILE, NEXT, RECORD, and REST clauses. If you look at the STD.CH header file, you will see that the COUNT, SUM, and AVERAGE commands, as well as the iterator versions of DELETE, RECALL, and REPLACE, are all preprocessed into calls to DBEVAL(). For example, if you want to sum the field BALANCE for all records in your database, the following DBEVAL() would do the trick:

```
ntotal := 0
dbeval( { | | ntotal += balance} )
```

You could easily modify this to keep track of the highest balance:

```
use customer new alias cust
ntotal := nmax := 0
dbeval( { | | ntotal += cust->balance, ;
              nmax := max(nmax, cust->balance) } )
? "Total:  ", ntotal
? "Maximum:", nmax
```

<block> is the code block to evaluate for each database record. There are a plethora of optional parameters.

<for> and <while> are code blocks that correspond directly to the FOR and WHILE clauses. Basically, if you use either or both of these clauses, DBEVAL() will process records until the code blocks return false (.f.). The following code fragment expands upon the previous example, and tracks the total and highest balance for all customers in the state of California ("CA") WHILE the record pointer is less than 200.

```
use customer new alias cust
ntotal := nmax := 0
```

```
dbeval( { | | ntotal += cust->balance, ;
              nmax := max(nmax, cust->balance) } ),;
        { | | cust->state = "CA"}, { | | recno() < 200} )
? "Total:   ", ntotal
? "Maximum:", nmax
```

<next> and <record> are both numerics; <next> specifies how many records to process from the current record, and <record> specifies which record number to process. Let's take another look at that example, but this time process the next 100 records for customers in the state of Oregon.

```
use customer new alias cust
ntotal := nmax := 0
dbeval( { | | ntotal += cust->balance,;
              nmax := max(nmax, cust->balance) }, ;
        { | | cust->state == "OR"},, 100)
? "Total:   ", ntotal
? "Maximum:", nmax
```

<rest> is a logical that determines whether the DBEVAL() scope will be from the current record to the end-of-file, or all records. If you pass true (.t.), DBEVAL() will assume that you prefer the former (i.e., start from current record). If you pass false or ignore this parameter, DBEVAL() will process all records.

Return Value: DBEVAL() always returns NIL.

ASCAN(<array>, <value>, [<start>], [<count>])
As in the Summer '87 version of Clipper, ASCAN() scans an array for a given <value>. However, the big difference is that you can now pass a code block as the <value>! "Why would I want to do that?" you moan. One useful situation would be the case-insensitive ASCAN(). First, try writing it in Summer '87. Then after you give up, bask in the comfort of knowing that it takes nothing more than a well-placed code block in Clipper 5:

```
ascan(myarray, { | a | upper(a) = upper("search value")} )
```

This will scan **myarray** and test the upper-case equivalent of each array element against the upper-case search value. But before we move on, let's bulletproof this code block. Do you know what happens if you try to convert a non-character value with UPPER()? (The answer is...an unexpected DOS holiday.) So let us ensure that each element thus tested is indeed a character string:

```
ascan(array, { | a | if(valtype(a) == "C", ;
                      upper(a) == upper(value), .f.) } )
```

An ounce of prevention is worth a day of debugging.

ASORT(<array>, [<start>], [<count>], [<block>])

As in Summer '87, ASORT() sorts an array. The optional parameters <start> and <count> are the same here as in AEVAL(). However, as with ASORT(), code blocks let you dramatically change the shape of things. You could come up with any manner of arcane sorts: Put all elements containing a certain substring at the top of the array; descending order; alphabetical order based on the last letter in the word (!). Each time your code block is evaluated by ASORT(), the function passes two array elements to the block. The block is then expected to compare them in some fashion that you specify, and return either true (.t.) if the elements are in proper order or false (.f.) if they are not.

Here's a descending sort:

```
local myarray := {"CHARLES", "JUSTIN", "JENNIFER", "TRACI", "DON"}
asort(myarray,,, { | x, y | x > y } )
aeval(myarray, { | a | qout(a) } ) // so you can see it worked!
```

One situation where a code block sort would save the day is when you must sort a multi-dimensional array. Let's fill an array with information from DIRECTORY(), and then sort it by filename. Bear in mind that DIRECTORY() returns an array containing one array for each file. The structure of this array is shown in Table 8.1.

Table 8.1 Structure of directory array

Array Element	*Information*	*Manifest Constant in DIRECTRY.CH*
1	file name	F_NAME
2	file size	F_SIZE
3	file date	F_DATE
4	file time	F_TIME
5	attribute	F_ATTR

(Refer back to Chapter 7, "The Preprocessor" if you need more information about manifest constants.)

In Summer '87, you must rely upon the soon-to-be-put-out-to-pasture ADIR() function, which requires that you establish an array for each piece of information that you wish to capture. Listing 8.2 compares the Summer '87 and Clipper 5 methods for sorting a directory by filename.

Listing 8.2 Sorting a directory by flename

```
* first in Summer '87
private files_[adir("*.*")]
adir("*.*", files_)
asort(files_)

* then in Clipper 5
#include "DIRECTRY.CH'
local files_ := directory("*.*")
asort(files_,,, { | x, y | x[F_NAME] < y[F_NAME] } )
```

Now let's sort the directory by date, as shown in Listing 8.3.

Listing 8.3 Sorting a directory by date

```
* Summer '87
private files_[adir("*.*")], dates_[adir("*.*")]
adir("*.*", files_, "", dates_)
asort(dates_)

* Clipper 5
#include "DIRECTRY.CH"
local files_ := directory("*.*")
asort(files_,,, { | x, y | x[F_DATE] < y[F_DATE] } )
```

You can see that the Summer '87 code has become increasingly convoluted as you add arrays to capture the other information. Not only that, but when you sort the **dates_** array, the **files_** array (which contains the filenames) is left unchanged, thus undercutting your best efforts. By stark contrast, you only need to change two digits in the Clipper 5 code, and do not have to worry about sorting one array while leaving another untouched. Now let's sort the directory by date and name, as shown in Listing 8.4.

Listing 8.4 Sorting a directory by date and name

```
* Summer '87
* I give up!

* Clipper 5
#include "DIRECTRY.CH"
local files_ := directory("*.*")
asort(files_,,, { | x, y | if( x[F_DATE] == y[F_DATE],;
                                x[F_NAME] < y[F_NAME], ;
                                x[F_DATE] < y[F_DATE] ) } )
aeval(files_, { | a | qout(padr(a[F_NAME], 14), a[F_DATE]) } )
```

(Note the use of PADR() to ensure that all the filenames line up.)

Because of the wonderful DIRECTORY() function, you can easily determine if the dates are the same (x[F_DATE] == y[F_DATE]). If they are, then compare the file names (x[F_NAME] < y[F_NAME]). Otherwise, compare the file dates (x[F_DATE] < y[F_DATE]). This is but one small example of something that you can now do in Clipper 5 that you could not (or would not dare) do in prior versions of Clipper.

If you want to sort by file extension, Clipper 5 makes it easy, as demonstrated in Listing 8.5.

Listing 8.5 Sorting a directory by file extension

```
#include "DIRECTRY.CH"
#translate ext( <f> ) => ;
   if('.' $ <f>, substr(<f>, at('.', <f>) + 1), '')

function main
local myarray_ := directory("*.*")
asort(myarray_,,, { | x, y | ext(x[F_NAME]) < ext(y[F_NAME]) } )
aeval(myarray_, { | a | qout(padr(a[F_NAME], 14), a[F_DATE]) } )
return nil
```

For the grand finale, let's sort the directory by date, file extension, and file name in Listing 8.6.

Listing 8.6 Sorting a directory by date, extension, and name

```
#include "DIRECTRY.CH"
#translate ext( <f> ) => ;
   if('.' $ <f>, substr(<f>, at('.', <f>) + 1), '')


function main
local myarray_ := directory("*.*")
```

```
asort(myarray_,,, { | x, y | if( x[F_DATE] == y[F_DATE],;
                              if( ext(x[F_NAME]) == ext(y[F_NAME]),;
                                      x[F_NAME] < y[F_NAME],;
                                  ext(x[F_NAME]) < ext(y[F_NAME])),;
                                      x[F_DATE] < y[F_DATE] ) } )
aeval(myarray_, { | a | qout(padr(a[F_NAME], 14), a[F_DATE]) } )
return nil
```

SETKEY(<key>, [<block>])

If you have ever used the Summer '87 SET KEY command to establish "hotkey" procedures, you may have been frustrated at the clumsiness of managing your hotkeys. For example, if you wanted to turn off all hot keys while the user was in a hotkey procedure, it required a certain degree of tedious coding. Hot key procedures are yet another area where Clipper 5 gives you unprecedented control. Whenever you establish a hotkey procedure with the SET KEY command, you are basically attaching a code block to that keypress with the new SETKEY() function. SETKEY() allows you to poll any INKEY() value to determine whether a code block is attached to it. Like all of the other SET() functions, it also permits you to change the current setting, i.e., attach a code block to any key.

The <key> parameter is a numeric corresponding to the INKEY() value of the keypress. (Please refer to Appendix A or the INKEY.CH header file for a complete listing of INKEY() values.)

The optional <block> parameter is the code block to be evaluated if the <key> is pressed during a wait state. Wait states include ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ, WAIT, and MENU TO. (See below for a discussion on INKEY(), the black sheep of the wait state family.)

SETKEY() either returns a code block if one is tied to the <key>, or nil. If you pass the <block> parameter, it will attach that code block to the <key>.

The SET KEY command

Before looking at any SETKEY() examples, let us first look at how the SET KEY command is handled in Clipper 5:

```
set key 28 to helpdev
```

gets translated by the preprocessor into the following:

```
setkey( 28, {|p, l, v| helpdev(p, l, v)} )
```

The **p**, **l**, and **v** parameters correspond to PROCNAME() (procedure name), PROCLINE() (current source code line number), and READVAR() (variable name), which will automatically be passed to the code block when it is evaluated. (Yes indeed, these are the same parameters passed to hot key procedures in Summer '87.) However, you can omit these arguments in your code block declaration if you will not be using them therein. By the same token, you are completely free to pass entirely different parameters to the function. (We will use this technique to pass local variables via code blocks a bit later.)

Whenever you come to a Clipper wait state, your keypress will be evaluated in approximately this fashion to determine whether or not there is a hot-key procedure tied to it:

```
keypress := inkey(0)
if setkey(keypress) != nil
   eval(setkey(keypress))
endif
```

SETKEY() == Better Housekeeping

Here is a good example where SETKEY() makes the difference between finding a solution and a gnashing of teeth: Suppose that within a hot key procedure you wish to temporarily attach a hot key definition to the F10 keypress. However, you may have F10 activating various different procedures throughout the course of your program. In Summer '87, this presented a big problem because you were unable to determine which procedure was tied to F10. You would therefore be unable to change it and expect to reset it properly. This is no longer a problem with SETKEY(). In Listing 8.7, we redefine F10 to call BlahBlah(), and reset it when we are finished.

Listing 8.7 SETKEY() Housekeeping

```
#include "inkey.ch"     // for INKEY() constants

function Test(p, l, v)
local old_f10 := setkey(K_F10, { | p,l,v | BlahBlah(p, l, v)} )
* main code goes here
setkey(K_F10, old_f10)                  // restore F10 hot key
return nil
```

old_f10 is assigned the code block (if any) that is attached to F10. F10 is then reassigned to trigger BlahBlah(). When we prepare to exit, we reset the previous code block (stored in **old_f10**) to the F10 keypress. (Once again, please remember that you can omit the **p**, **l**, **v** arguments in your code block declaration if you will not be using them in the hotkey function.)

Important Note: Before you go hog wild with hot keys, you should know that there are a limit of 32 SETKEY() (or SET KEY, same difference) procedures at any given time.

INKEY() := Wait State?

As with Summer '87, INKEY() is not a bona fide wait state. But as you have just seen, SETKEY() makes it very easy to create your own INKEY() wait state. Listing 8.8 shows the function Ginkey() from the Grumpfish Library.

You may have already noticed that there is a function similar to this in the file KEYBOARD.PRG, which is included on your Clipper 5 distribution diskettes. However, the Nantucket function does not take into account a strange quirk of the Clipper 5 INKEY(). If you explicitly pass a NIL to INKEY(), the function will perform as though you passed it zero — it will wait indefinitely for a keypress.

Therefore, the INKEY() wait state function must be smart enough to differentiate between a NIL passed to it, and a NIL that comes as a result of NOT receiving a formally declared parameter. (For more information about NIL, please review Chapter 4, "Data Types".)

Special thanks to Jeff Gruber for bringing this INKEY() idiosyncrasy to light.

Listing 8.8 INKEY() as a wait state

```
/*
      GINKEY() - INKEY() wait state
      Excerpted from GRUMPFISH LIBRARY
      Author: Greg Lief
*/
function Ginkey(waittime)
local key, cblock
do case
   /* if no WAITTIME passed, go straight through */
   case pcount() == 0
        key := inkey()
   /* if you pass inkey(nil), it is identical to INKEY(0) */
   case waittime == nil .and. pcount() == 1
        key := inkey(0)
   otherwise
        key := inkey(waittime)
endcase
cblock := setkey(key)
if cblock != nil
   // run the code block associated with this key and pass
   // name of previous procedure and previous line number
   eval(cblock, procname(1), procline(1), 'Ginkey')
endif
return key
```

As mentioned earlier, the third parameter passed to hot key procedures is the name of the variable being read. In this function, "Ginkey" is serving as a dummy variable name. Please feel free to change it to anything you desire. If you really wanted to, you could pass a variable name as a second parameter to Ginkey(), and in turn pass that to the code block if and when it was evaluated.

Notice that when the code block is evaluated, instead of passing it the current procedure name and line number, we pass it the information that is one level previous on the activation stack. Otherwise, the hot key procedure would always think that it had just come from GINKEY(). This would in turn louse things up by forcing you to have the same help screen for every GINKEY() wait state.

FIELDBLOCK(<field>)

FIELDBLOCK() is the first of three new functions that return "set-get" code blocks. One of the biggest reasons to use this trio of functions is to preclude the use of the macro operator. (As you should have already surmised from the way we carry on about them, swearing off macros will make your programs run faster and look more svelte.)

FIELDBLOCK() returns a retrieval/assignment code block for a specified field. The parameter <field> is a character string representing the field name to refer to. You can then either retrieve (get) or assign (set) the value of <field> by evaluating the code block returned by FIELDBLOCK(). If <field> does not exist in the currently active work area, FIELDBLOCK() will return NIL.

Note: if the <field> that you pass to FIELDBLOCK() exists in more than one work area, FIELDBLOCK()'s return value will correspond only to the <field> in the current area. Here's an example of retrieving the value:

```
local bblock, mfield := "FNAME"
dbcreate("customer", { { "FNAME", "C", 10, 0 } })
use customer new
append blank
customer->fname := "JOE"
bblock := fieldblock(mfield)
? eval(bblock)      // "JOE"
```

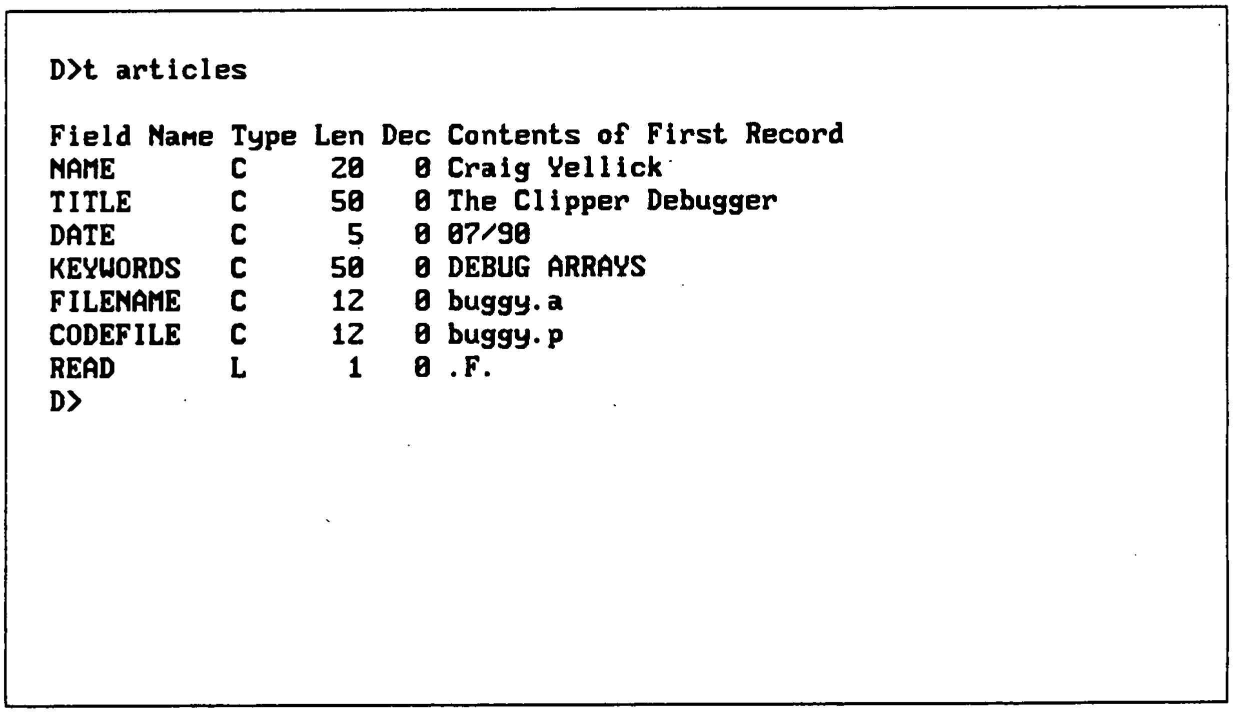
```
/* note the dreaded macro alternative */
? &mfield          // slow, and simply no longer chic
```

To assign a value to a field, you merely evaluate the code block and pass the desired value as a parameter. For example:

```
local bblock, mfield := "FNAME"
use customer new
bblock := fieldblock(mfield)
eval(fieldblock(mfield), "Craig")
? customer->fname     // "Craig"
/* note the dreaded macro alternative */
replace &mfield with "Craig"  // ugh!
```

The function Struct() loops through the structure array created by DBSTRUCT() and uses FIELDBLOCK() to retrieve the value for each field in your database. Listing 8.10 contains the source code for Struct(), and Figure 8.1 shows sample output.

Figure 8.1 Sample output from function Struct()



```
D>t articles

Field Name Type Len Dec Contents of First Record
NAME       C     20   0 Craig Yellick
TITLE      C     50   0 The Clipper Debugger
DATE       C      5   0 07/90
KEYWORDS   C     50   0 DEBUG ARRAYS
FILENAME   C     12   0 buggy.a
CODEFILE   C     12   0 buggy.p
READ       L      1   0 .F.
D>
```

Listing 8.10 Showing .DBF structure with FIELDBLOCK()

```
function Struct(dbf_file)
local struct, x
if dbf_file == NIL
   qout("Syntax: struct <dbf_name>")
elseif ! file(dbf_file) .and. ! file(dbf_file + ".dbf")
   qout("Could not open " + dbf_file)
else
   use (dbf_file) new
   struct := dbstruct()
   qout("Field Name Type Len Dec Contents of First Record")
   for x = 1 to len(struct)
       qout(padr(struct[x, 1], 10), padr(struct[x, 2], 4), ;
            str(struct[x, 3], 3), str(struct[x, 4], 3),    ;
            eval(fieldblock(struct[x, 1])) )
   next
   use
endif
return nil
```

FIELDWBLOCK(<field>, <work area>)

FIELDWBLOCK() is quite similar to FIELDBLOCK(). However, as you may have already guessed from the "W" in its name, it allows you to refer to a different work area to retrieve or assign the <field> value.

As with FIELDBLOCK(), the <field> parameter is a character string representing the field name to refer to. The new parameter <work area> is a numeric indicating which work area to search for the <field>.

Once again, you can then either retrieve or assign the value of <field> by evaluating the code block returned by FIELDWBLOCK(). If <field> does not exist in the specified <work area>, FIELDWBLOCK() will return NIL. (Note: FIELDWBLOCK() does not change the active work area.)

Here's FIELDWBLOCK() in action. Note the use of the SELECT() function to determine the work areas; this is much easier than having to hard-code (and then remember) work area numbers.

```
dbcreate("customer", { { "LNAME", "C", 10, 0 } })
dbcreate("", { { "LNAME", "C", 10, 0 } })
use customer new
append blank
customer->lname := "CUSTOMER1"
use vendor new
append blank
vendor->lname := "VENDOR1"
? eval(fieldwblock("LNAME", select("customer"))) // CUSTOMER1
? eval(fieldwblock("LNAME", select("vendor")))   // VENDOR1
? eval(fieldwblock("LNAME", select("vendor")), "Grumpfish")
? vendor->Lname                                  // Grumpfish
```

As with FIELDBLOCK(), it is quite easy to assign a value to a field. Simply evaluate the code block returned by FIELDWBLOCK() and pass the desired value as a parameter. We used this method to change the field LNAME in VENDOR.DBF in the next to last line above.

Earlier in this chapter, we showed an example of creating a generic TBrowse object to browse a database. FIELDWBLOCK() offers a different solution:

```
local x, browse := TBrowseDB(3, 19, 15, 60), column
use test new
for x := 1 to fcount()
   column := TBColumnNew(field(x),;
                         fieldwblock(field(x), select()))
   browse:AddColumn( column )
```

```
next
do while ! browse:stabilize()
enddo
```

MEMVARBLOCK(<memvar>)

MEMVARBLOCK() is also quite similar to FIELDBLOCK(), except that it operates upon memory variables rather than database fields. MEMVARBLOCK() returns a code block for a memory variable as specified by the <memvar> parameter. You can then either retrieve the value of <memvar> by evaluating the code block returned by MEMVARBLOCK(), or assign <memvar> a value by evaluating the code block and passing the value as a parameter.

If the <memvar> does not exist, MEMVARBLOCK() will return NIL.

Very Important Note: If the <memvar> is either static or local, MEMVARBLOCK() will also return NIL. This is because MEMVARBLOCK() can only operate on public and private variables. In this example, MEMVARBLOCK() retrieves the value of each of four memory variables.

```
// note PRIVATE declaration - MEMVARBLOCK() doesn't like locals

private mtot1 := 75, mtot2 := 400, mtot3 := 30, mtot4 := 205, x
for x := 1 to 4
   ? eval(memvarblock("mtot" + str(x, 1)))
next
```

FIELDGET(), FIELDPUT(), FIELDPOS()

Although these functions do not use code blocks, they are conceptually related to FIELDBLOCK(). All of these functions make it very easy for us to create generic "scatter" and "gather" routines without the use of macros. "Scatter" routines dump database fields to memory variables for editing, and "gather" routines assign values in said memory variables to database fields.

FIELDGET() and FIELDPUT() accept one numeric parameter, which corresponds to a field's ordinal position in the database structure. FIELDPUT() also accepts a second parameter, which is the value to assign to the field.

Both of these functions return the value of the field in question. If the field number parameter does not correspond with any fields in the database structure, they will return NIL.

Listing 8.11 demonstrates two scatter/gather routines, one with macros, and one with FIELDGET()/FIELDPUT().

Listing 8.11 Scatter/Gather with FIELDGET()/FIELDPUT() and macros

```
function Test
local nfields, xx, ahold := {}, mfield
memvar getlist
// first create test database
dbcreate('rolodex', { { "FNAME",  "C", 15, 0}, ;
                      { "LNAME",  "C", 15, 0}, ;
                      { "ADDRESS","C", 35, 0}, ;
                      { "CITY",   "C", 30, 0}, ;
                      { "STATE",  "C", 2,  0}, ;
                      { "ZIP",    "C", 10, 0} } )
use rolodex new
nfields := fcount()
cls
/* first let's try it with macros */
// dump all field contents to array for editing
for xx := 1 to nfields
   mfield := field(xx)
   aadd(ahold, &mfield)          // the macro, aarrgghhhh
   @ xx, 1 say padr(mfield, 11) get ahold[xx]
next
read
append blank
```

```
// now dump array contents to the fields of the blank record
for xx := 1 to nfields
   mfield := field(xx)
   replace &mfield with ahold[xx]   // the macro again, ugh!
next

/* now with FIELDGET() and FIELDPUT() */
ahold := {}   // clear out the array
// dump all field contents to array for editing
for xx := 1 to nfields
   aadd(ahold, fieldget(xx))        // look ma, no macro
   @ xx, 1 say padr(field(xx), 11) get ahold[xx]
next
read
// now dump array contents to the fields of this record
aeval(ahold, { | ele, num | FieldPut(num, ele) } )
return nil
```

This example is fairly simple, because it creates the test database each time. However, suppose that you want to be able to add a record to a database that already has records. Rather than going through gyrations to determine the initial values of each field, you can simply issue a "GO 0" command prior to loading the **ahold** array.

Any attempt to GO to a record that is out of range will position the record pointer at LASTREC() + 1, which is sometimes referred to as "the phantom record."

In the above example, we scattered and gathered all of the fields in the database. However, suppose that you only wanted to scatter/gather two of the fields. That is where the FIELDPOS() function (new with Clipper 5.01) comes into play. FIELDPOS() returns the position of a specified field within the DBF structure corresponding to the active work area.

The syntax for FIELDPOS() is:

```
FIELDPOS( <cField> )
```

<cField> is the name of the desired field in the current work area. FIELDPOS() returns the ordinal position of the specified field in the .DBF structure associated with the current work area. If the field cannot be found in the current work area, FIELDPOS() returns zero.

Listing 8.12 relies upon FIELDPOS() to scatter/gather two of the fields from our ROLODEX database.

Listing 8.12 FIELDPOS() for scatter/gather

```
function Main
local fields_ := { "LNAME", "ADDRESS" }, ahold_ := {}
use rolodex new
for x := 1 to len(fields_)
  aadd(ahold_, fieldget(fieldpos(fields_[x])) )
  @ x, 1 say padr(fields_[x] + ":", 12) get ahold_[x]
next
read
// now dump array contents to the fields of this record
aeval(ahold_, { | ele, num |;
     fieldput(fieldpos(fields_[num]), ele) } )
return nil
```

Tying local variables to code blocks

We all know that the scope of a **local** variable is the procedure or function in which it is declared. However, you can access local variables belonging to lower-level functions. To do so, you must create a code block in the lower-level function that refers to the local variable, and pass that block back to the higher-level function. The following code snippet demonstrates this principle:

```
function Test
local myblock
myblock := ClipVer()
? eval(myblock)
return nil
```

```
static function ClipVer
local xx := "Clipper 5.01"
return { | | xx }
```

The variable **xx** remains accessible via the code block even though the function ClipVer() is no longer active. The reference to **xx** remains active as long as any code blocks referring to it (such as **myblock**) remain active.

This ability has tremendous ramifications for code blocks. One problem it solves is where a code block needs long-term ownership of certain values. For example, you should only need to macro-compile a code block if the actual code to be executed isn't known until run-time, as opposed to situations where the code block must contain a value which isn't known until run-time (this would have traditionally been handled with the macro operator).

As always, each time you call a function, a unique set of local variables is generated. This rule applies even when using the aforementioned technique with code blocks. Take the following code fragment:

```
function Test
local myblock1, myblock2
myblock1 := Counter()
myblock2 := Counter()
? eval(myblock1)   // output: 1
? eval(myblock1)   // output: 2
? eval(myblock2)   // output: 1
return nil

static function Counter()
local xx := 1
return { | | xx++ }
```

myblock1 and **myblock2** both contain references to the variable **xx**, but they are completely independent of each other. This is demonstrated by evaluating **myblock1** twice, which increments its copy of **xx** to 2. However, evaluation of **myblock2** reveals that its copy of **xx** still contains the value of 1.

Passing local variables in a code block

In similar fashion, you can actually pass **local** variables to other functions via code blocks. Watch this:

```
function Main
local bblock := { | | x }, x := 500
Test1(bblock)
return nil

function Test1(b)
? eval(b)   // output: 500
return nil
```

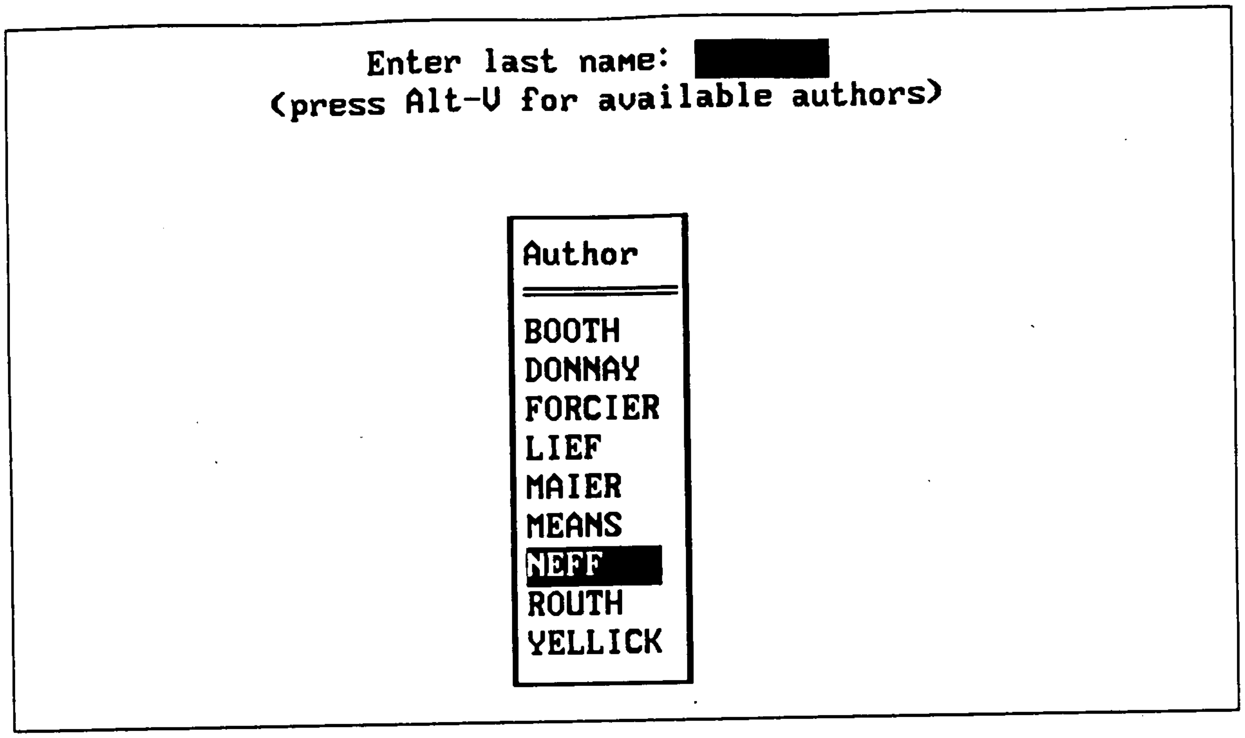
When **bblock** is compiled in Main(), it will contain a reference to **x**, which is a variable local to Main(). However, when the block **bblock** is passed as a parameter to Test1(), and subsequently evaluated therein, **x**'s value will indeed be available.

Note: We do not advocate the unmitigated use of this technique. It does not seem to be exactly what the Clipper 5 architects had in mind for local variables. But one situation comes to mind where this method would save the day.

You need to GET a variable and wish to allow the user to press a hot key to pop up a list of valid entries. This sounds pretty simple, doesn't it? It would be, except that the GET variable in question is local and thus restricted in scope to the function in which we are getting it. What to do... what to do?

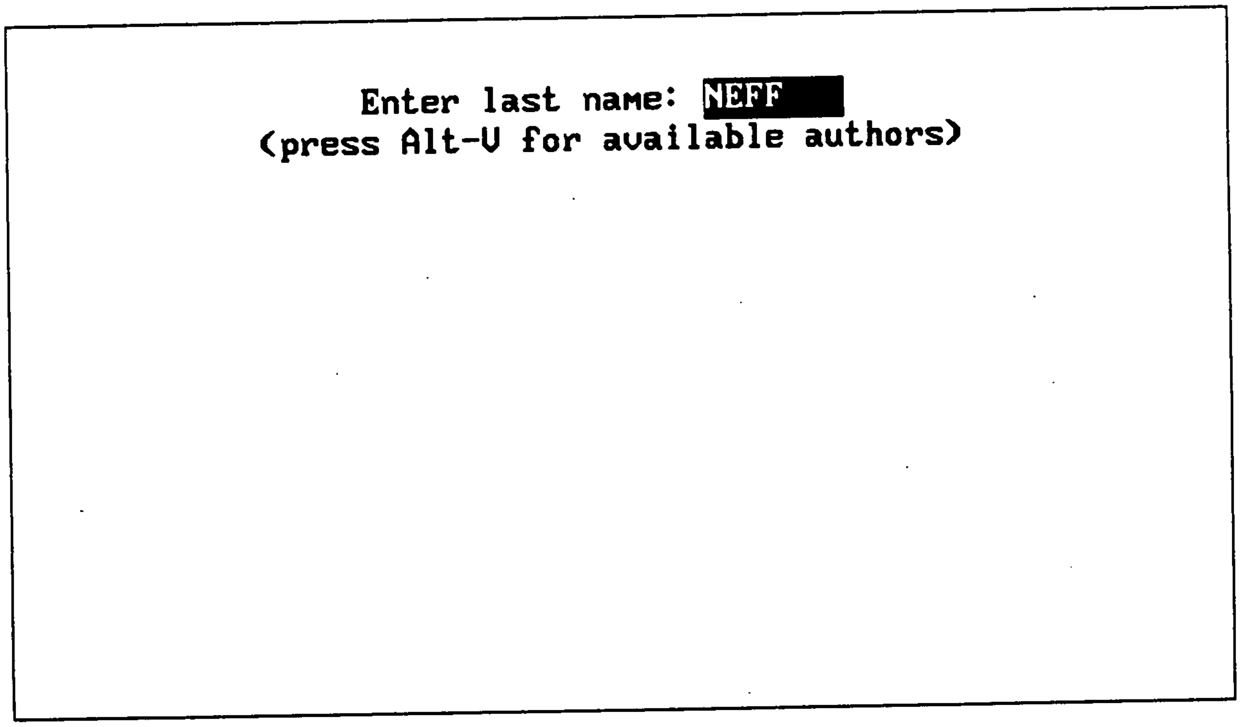
Listing 8.13 shows code that solves the problem with the clever use of a code block. Figure 8.2 shows the pop-up picklist on the screen, with the name "Neff" highlighted for selection. Figure 8.3 proves conclusively that this value gets properly assigned to the local variable in the calling routine.

Figure 8.2 Pop-up pick list of author names



```
          Enter last name:
    (press Alt-U for available authors)



                    Author
                    ══════
                    BOOTH
                    DONNAY
                    FORCIER
                    LIEF
                    MAIER
                    MEANS
                    NEFF
                    ROUTH
                    YELLICK
```

Figure 8.3 GET changed via code block



```
          Enter last name: NEFF
    (press Alt-U for available authors)
```

Listing 8.13 Passing local variables in a code block

```
#include "inkey.ch"
#include "box.ch"


function Test
local mvalue := space(7), oldaltv, x
memvar getlist
if ! file("lookup.dbf")
   dbcreate("lookup", { { "LNAME", "C", 7, 0 } } )
   use lookup new
   for x := 1 to 9
      append blank
      /* note use of unnamed array - it works just fine this way */
      replace lookup->lname with { "BOOTH", "DONNAY", "FORCIER", ;
                 "LIEF", "MAIER", "MEANS", "NEFF", "ROUTH", ;
                 "YELLICK" }[x]
   next
else
   use lookup new
endif
/* note that we pass MVALUE by reference to VIEW_VALS() below */
oldaltv := setkey( K_ALT_V, {| | View_Vals(@mvalue)} )
cls
@ 4, 28 say "Enter last name:" get mvalue
@ 5, 23 say '(press Alt-V for available authors)' color '+w/b'
read
quit

static function View_Vals(v)
local browse, column, key, marker := recno(), ;
      oldscrn  := savescreen(8, 35, 20, 44, 2), ;
      oldcolor := setcolor("+W/RB"), oldcursor := setcursor(0), ;
      oldblock := setkey( K_ALT_V, NIL)  // turn off ALT-V
@ 8, 35, 20, 44 box B_SINGLE + chr(32)
browse := TBrowseDB(9, 36, 19, 43)
browse:headSep := "-"
browse:colorSpec := '+W/RB, +W/N'
```

```
column := TBColumnNew( "Author", FieldBlock("lname") )
browse:addColumn(column)
go top
do while .t.
   do while ! browse:stabilize() .and. (key := inkey()) == 0
   enddo
   if browse:stable
      key := inkey(0)
   endif
   do case
   case key == K_UP
      browse:up()
   case key == K_DOWN
      browse:down()
   case key == K_ESC .or. key == K_ENTER
      exit
   endcase
enddo
if lastkey() != K_ESC
   /*
    because we passed the variable BY REFERENCE in the code
    block, any changes we make here are being made to the actual
    variable, and that is the key to this whole mess working the
    way it does!
   */
   v := eval(fieldblock('lname'))
endif
go marker
restscreen(8, 35, 20, 44, oldscrn)
setcolor(oldcolor)
setcursor(oldcursor)
setkey(K_ALT_V, oldblock)   // reset Alt-V for next time
return nil
```

Summary

Any mental blocks that you formerly had about code blocks should be lying in pieces around your chair. You should now be absolutely fearless in the presence of code blocks. In fact, you might even like them. You are now certainly able to write your own. You understand how to pass parameters to code blocks, and how to evaluate them. You are also familiar with the many Clipper 5 functions that use them.

Before you go on to any other chapters, move over to the PC and experiment with code blocks. Test out the examples in this chapter (which are all available on the accompanying source code diskette). Write a few of your own. Go wild! As with most things in Clipper 5, your imagination should be your only limit when dealing with code blocks. After you have written several dozen of your own, turn off the computer and take a break. You've earned it!

CHAPTER 9

Arrays

Arrays are a powerful feature of the Clipper language and are implemented very differently than arrays in other languages. Arrays in Clipper Summer '87 are relatively crude compared to the power and flexibility offered in Clipper 5. Additionally, arrays have become a fundamental part of the Clipper 5 language extensions rather than being a minor feature of Summer '87. To get the most out of Clipper you need to be comfortable with arrays.

This chapter starts with an introduction to arrays as they appear in Summer '87, then describes the major enhancements found in Clipper 5 and offers simple examples. We then discuss how arrays can be put to effective use in more complex situations.

After reading the first half of this chapter you should understand the major features of Clipper arrays and know how to take advantage of their flexibility. The second half should generate ideas and motivate you to use arrays for more than just simple list processing. Don't try to read this chapter straight through from beginning to end. Try to get some practical experience with Clipper 5's new array capabilities before forging ahead into the applications section.

A Summer '87 introduction

Arrays in Clipper Summer '87 are considerably more limited and are conceptually easier to understand than in Clipper 5. This section introduces arrays from the context of their implementation in Summer '87. Everything you read in this section is completely applicable to Clipper 5. To reduce confusion we will use Summer '87 syntax in the source code examples. Keep in mind that this is a simplified introduction and you should not pattern your use of arrays after the syntax shown here; wait until we discuss the substantial improvements found in Clipper 5.

Arrays are single memory variable names that can hold more than a single value, similar to the way a single database file name can hold more than a single record. Arrays must be declared as such prior to making assignments. In Summer '87 you need to specify, in advance, the maximum number of values the array name can hold. This maximum number is referred to as the array's size or length, and the individual values are called elements. The declare statement is used to create an array and establish the maximum number of elements.

```
declare aList[100]
```

In the above example an array called aList is created with a maximum of 100 elements, meaning aList can contain between zero and 100 different values.

The maximum size of any array is 4,096 elements. Each element is accessed by specifying its element number. Element numbers are like record numbers, starting with element number one. This is different from other languages which start numbering elements with zero.

The following assigns a character string to the first element in the aList array.

```
aList[1]= "The first element in the array"
```

Another difference between Clipper arrays and arrays found in other languages is that Clipper allows any combination of variable types in the same array. Array elements start out as undefined, and can be assigned and changed at any time. We'll be referring to the following example for the next several paragraphs.

```
declare aList[5]
aList[1]= "XYZ"
aList[2]= 987
aList[3]= .t.
aList[4]= date()
```

In the above example, aList has a maximum of five elements, the first four of which are assigned values of different data types. The following commands will yield the indicated results.

```
? type("aList")     &&  "A" for array
? type("aList[1]")  &&  "C" for character
? type("aList[2]")  &&  "N" for number
? type("aList[3]")  &&  "L" for logical
? type("aList[4]")  &&  "D" for date
? type("aList[5]")  &&  "U" for undefined
```

Programmers working in "pure" languages see this as a cursed abomination. We see it as a wonderful expression of the blind trust that our language places in the programmer.

Important point: Note how the TYPE() function returns an "A" when just the array name is passed, while returning the appropriate type of individual array elements when specific elements are passed. In Clipper 5 use of the VALTYPE() function is preferred over TYPE(). TYPE() uses the macro operator to determine the data type and consequently does not work with static and local variables. See Chapter 4, "Data Types," for details.

An array element can be used as a parameter for any function or procedure, just like a regular memory variable. In addition to TYPE(), the LEN() function also has a dual nature when working with arrays.

```
? len(aList)     &&  5, the number of elements in aList
? len(aList[1])  &&  3, the length of "XYZ"
? len(aList[3])  &&  run-time error, type mismatch
```

Another important point: Note the LEN() function returns the number of elements declared for the array, which is not necessarily the number of elements with useful values. When a specific array element is passed to LEN(), the expected type checking is in place. In the example, element number three contains a logical value, resulting in LEN() taking a side trip to the run-time error handler.

To refer to the entire array you simply use the array name without an element number. In this example, the entire aList array is passed as a parameter to the ACHOICE() function.

```
achoice(5,10,12,22, aList)
```

Unlike regular memory variables, arrays are always passed to functions by reference, not value. This means that any changes made to the array by the function will be reflected in the calling routine.

```
MyFunc(aList)
quit

function MyFunc
parameter a
private i
for i = 1 to len(a)
  a[i] = i
next i
return "."
```

In the above example, the **aList** array is modified by MyFunc(). There is no way to pass an entire array by value to a function. You can, however, pass *individual elements* by value to functions.

```
AnotherFunc(aList)         && Pass entire array by reference
AnotherFunc(aList[1])      && Pass element by value
```

In the above example, **aList** is passed by reference, so any changes made in AnotherFunc() will be reflected in **aList**. The reference to **aList[1]** is for only a single element, so only the value of the array element is passed to the function. Changes made within AnotherFunc() are not reflected in the **aList[1]** array element.

Summer '87 arrays cannot shrink or grow in size once they are declared to be a specific length. Memory occupied by an array is not released until the program exits the routine where the array was declared. Arrays can be declared to be **private** or **public**, with the same effect as for regular memory variables. The DECLARE statement is a synonym for **private**.

```
public acolors[15]
private adata[250]
```

If you assign a memory variable the same name as an existing array, the entire array is wiped out and replaced by the single value.

```
declare anames[10]      && Ten elements in the array
afill(anames, "abc")    && Assign "abc" to all elements
anames= 123             && Array is wiped out
```

Before they had true arrays, xBASE programmers often simulated them with macros. While such a simulation was often the only way to accomplish many goals, the use of macros exacts a heavy price in speed and memory use.

```
* Before arrays: Result is regular memory variables
* named TEST_1 through TEST_100 being created.
*
for i = 1 to 100
  c = ltrim(str(i, 3))
  test_&c. = i
next i
```

```
* With arrays: Result is single name with 100 values,
* TEST_[1] through TEST_[100].
*
declare test_[100]
for i = 1 to 100
  test_[i] = i
next i
```

In Summer '87, only two functions use arrays as parameters for more than just manipulating the array. These functions perform user-interface actions: ACHOICE() and DBEDIT(). All other array-related functions are used to manipulate the contents of existing arrays: AFILL(), AINS(), ADEL(), AFIELDS(), and ADIR().

An Alternate Naming Convention

Nantucket's documentation uses a naming convention that indicates a variable's data type via the first letter of the variable's name.

```
iNumber := 100      // Integer
cString := "abc"    // Character
aArray := {1,2,3}   // Array
//  Etc
```

While this is very useful for keeping things straight in general, we have developed a variation on this convention that helps even more with arrays. Our convention is to place a trailing underscore character at the end of the array's variable name. The leading letter convention can then be used to indicate the contents of the array rather than the mere fact that the variable is an array.

```
iEven_     := {2,4,6,8}
cState_    := {"MN","OR","NY"}
lChosen_   := {.t., .t., .f., .f.}
```

We are not as persnickety about always using the first letter of the variable name to indicate the type. We usually only use the convention when the variable's type is not immediately obvious in the context it appears. So, in the remainder of this chapter when you see a trailing underscore you know the variable is an array. If no leading character is used it implies that the data type does not matter for the purpose of the example source code, or that the array contains a mixture of data types, or even that the contents are arbitrary.

A helpful side effect of the trailing underscore is to visually separate the name of the array from the element number in square brackets. In dense source code lines it is easier to see that a reference is being made to an array element instead of a function call with parameters.

```
A := MyFunc(str(z), this[1] + that(2))
B := MyFunc(str(z), this_[1] + that(2))
```

The assignment to **A** as compared to **B** is slightly more clear, especially when the lines are buried among hundreds of lines of similar code or when viewed on a 43-line monitor or condensed source code listing. The purpose is to help you differentiate more easily between array references and function calls, since they differ only by the shape of the delimiters. In the above example, **this[1]** looks very similar to a function called **this(1)**, while **this_[1]** is less likely to be mistaken.

What's new in Clipper 5?

Clipper 5 expands Clipper's support of arrays in so many ways that it's hard to know where to begin. In Summer '87 arrays are essentially alternatives to memory variables and macros, or "memory databases" with one field. In Clipper 5 arrays have become a fundamental part of the language, implemented in a way that makes them more than simple alternatives to memory variables or databases.

Dynamic Size

Arrays no longer need to be declared in advance with a maximum size. You can start with an array of zero elements and add (or delete) elements as needed, at any time. This is distinct from the AINS() and ADEL() functions found in Summer '87. AINS() inserts an undefined value in an array and shifts the rest of the elements down, dropping any that go beyond the original declared length. ADEL() moves all elements up, placing an undefined element at the end of the array (or specified section of the array). Neither AINS() or ADEL() alter the actual *size* of the array, only the *contents*. The functions AADD() and ASIZE(), new in Clipper 5, are used to dynamically alter the size of arrays.

```
//  Array starts out with ten elements
local this_[10]
? len(this_)  //  10

//  Array now has eleven elements
aadd(this_, "abc")
? len(this_)  //  11
? this_[11]   //  "abc"

//  Array has been reduced to seven elements
asize(this_, 7)
? len(this_)  //  7

//  Array has been increased by 20 to 27 elements
asize(this_, len(this_) +20)
? len(this_)  //  27
```

Note the AADD() function's ability to supply a value to assign to a new element.

```
//  Add new element, value is "abc"
aadd(this_, "abc")

//  Add another new element, value is 123
aadd(this_, 123)
```

ASIZE() does not assign values. It either adds or removes elements to match the specified size.

Multiple dimensions

Any element in an array can be another array, nested as deeply as you wish. One dimension isn't terribly difficult to grasp: If you declare that **this_** has ten values and number each value, **this_**[5] is the fifth value in **this_**. Additional dimensions are often confusing to beginning programmers (and not just a few "advanced" ones), so an explanation is in order.

This section introduces multidimensional arrays by using a database file as an analogy. Not everyone will find this intuitively obvious. If not, consider jumping ahead to the section on practical applications of multidimensional, variable array structures which uses a different analogy.

When you work with a database you are using what amounts to a two-dimensional array. Let's use the datebase structure in figure 9.1 as an example. The first dimension is the record number: If THIS.DBF has ten records, each record is associated with a number between one and ten. The second dimension is the field: Each record in THIS.DBF has four fields; the second field is the customer name.

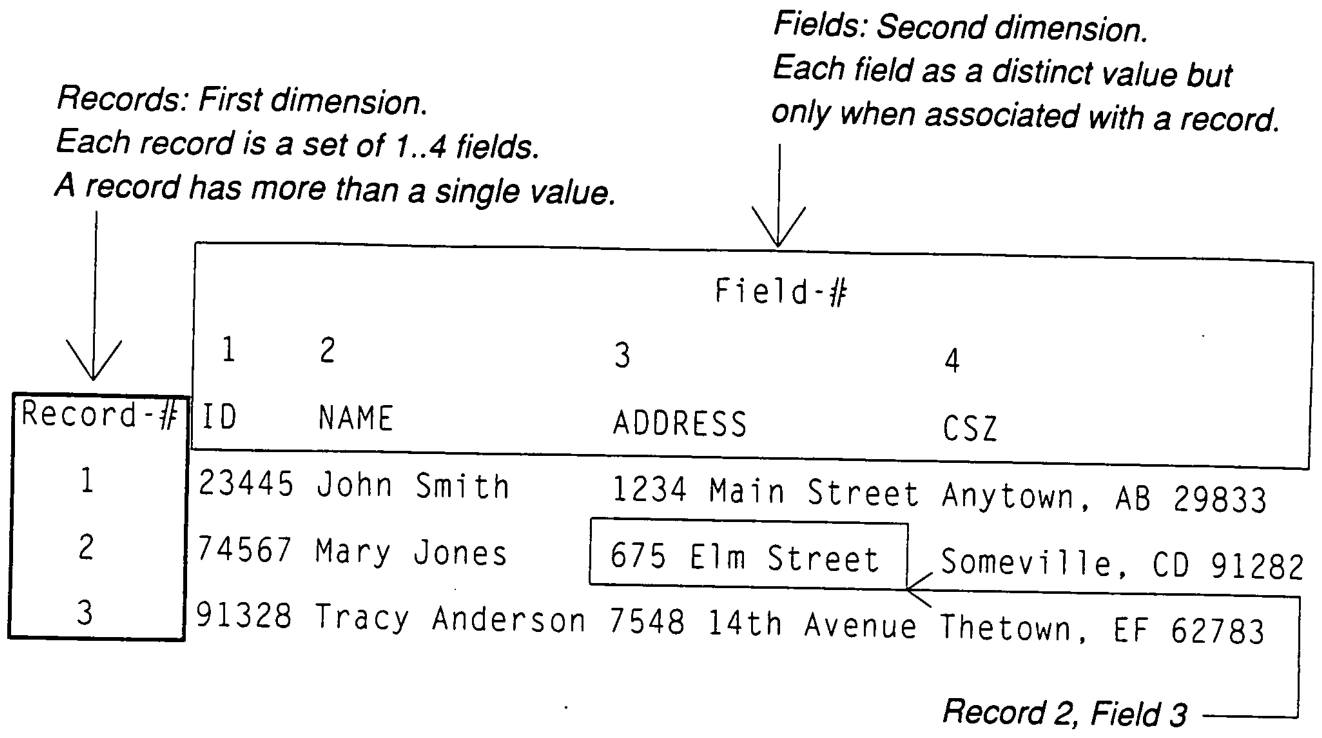
Figure 9.1 A simple database structure

THIS.DBF:

Field	*Name*	*Width*
1	ID	3
2	NAME	20
3	ADDRESS	20
4	CSZ	20

For the visually oriented reader, Figure 9.2 depicts the database analogy for arrays with multiple dimensions.

Figure 9.2 Seeing a database listing as a multi-dimensional array


Fields: Second dimension.
Each field as a distinct value but
only when associated with a record.
Records: First dimension.
Each record is a set of 1..4 fields.
A record has more than a single value.
Field-#
Record 2, Field 3

	Field-#			
	1	2	3	4
Record-#	ID	NAME	ADDRESS	CSZ
1	23445	John Smith	1234 Main Street	Anytown, AB 29833
2	74567	Mary Jones	675 Elm Street	Someville, CD 91282
3	91328	Tracy Anderson	7548 14th Avenue	Thetown, EF 62783

Record 2 has more than just a single value:

```
{"74567", "Mary Jones", "675 Elm Street", "Someville, CD 91282"}
```

Field 3 for record 2 has only a single value:

```
"675 Elm Street"
```

If THIS.DBF were an array named THIS, then THIS[2] refers to the entire contents of the second record. THIS[2,3] refers to the single value found in field three of the second record.

We can go to a desired record and refer to the field name of a desired field. We can, if we want to, refer to the fields by number instead of field name. We use field names because they make our code easier to read and maintain. Let's make up a user-defined function, called Go_get(), that returns a specified field from a specified record. It could look something like the following.

```
// Record 5, field 4
cust_name := Go_get(5, 4)

function Go_get(rec, fld)
goto (rec)
return fieldget(fld)
```

An array with two dimensions can be constructed and accessed in the same way. The first dimension is an array of record numbers. When you refer to an element in the first dimension you are talking about the complete set of fields for that record. You must supply an element number in the second dimension in order to refer to a specific value in a field. The following example illustrates this concept (assume the contents of a database are loaded into a two-dimensional array called DATA_).

```
full_record := data_[5]    // An array of field values
                           // for record 5
cust_name := data_[5, 4]   //  Value of field 4
```

Clipper allows two ways to refer to elements in an array with more than one dimension. You can list the element in each dimension with a pair of square brackets, or specify each individual element in its own brackets.

```
? data_[5, 4]
? data_[5][4]
```

Our preference is listing all dimensions together. All those square brackets make source code difficult to read, especially in arrays with many dimensions.

The preprocessor allows us to deal with multiple dimensions in a more understandable way by using the #define directive to assign names to the dimensions, as illustrated in the following example.

```
#define ID      1
#define NAME    2
#define ADDRESS 3
#define CSZ     4
```

```
? data_[3, NAME]    //  Field 2 in record 3
? data_[1, CSZ]     //  Field 4 in record 1
```

The manifest constants ID, NAME, ADDRESS, and CSZ make immediate sense to anyone reading the source code, compared to the more abstract numbers 1, 2, 3, and 4.

Ok, so two dimensions isn't all that hard to grasp when compared to database structures. Let's tackle a three-dimensional array. Once again, you are already familiar with a third dimension when using databases! Each field has a number of characters that form the value. It's easy to determine the seventh character of the second field of the fifth record of a database, right?

```
goto 5
? substr(THIS->Name, 7, 1)  //  "Name" is the 2nd field
```

Extending the user-defined function we discussed earlier, we could write something like the following.

```
//  Record 5, field 2, character 7
character_7 := Go_get(5, 2, 7)
```

If a database was loaded into an array called **data_**, the following references can be made.

```
full_record := data_[2]         //  Array of field arrays
                                //    for record #2
field_chars := data_[2, 3]      //  Array of characters
                                //    in field #3
a_character := data_[2, 3, 7]   //  7th character
                                //    in field #3
```

When using a multiple dimension array you must be careful to keep track of what you're referring to. The number of elements in each dimension is important, too. Use of preprocessor #define directives makes array references easier to understand. The ID and NAME fields are the first and second elements in the array.

```
#define ID    1
#define NAME  2
```

In the previous example, the ID field is three characters wide while the name is 30 characters wide. A reference to **data_[2, NAME, 25]** is legal because we are referring to the 25th element in an array of 30 elements. However, a reference to **data_[2, ID, 25]** is not legal because there are only three elements in that array. The following is an example of the ID field array being assigned "A,B,C" for record number two.

```
data_[2, ID, 1] := "A"
data_[2, ID, 2] := "B"
data_[2, ID, 3] := "C"
```

If we are using only two dimensions we write the following instead.

```
data_[2, ID] := "ABC"
```

If multiple array dimensions still isn't making sense, please hang in there. There are many more examples and explanations still to come.

Abbreviated syntax

Clipper makes array declarations and assignments considerably easier through the use of the inline assignment operator and the curly brace delimiters. The following are equivalent.

```
local iList_[3]
iList_[1] := 10
iList_[2] := 20
iList_[3] := 30
*
local iList_ := { 10, 20, 30 }
```

The array called **iList_** is termed a literal array, because it is created directly from the source code syntax and not from a run-time process.

Clipper's completely dynamic implmentation of arrays means you don't need to know the size of the array when it is first created.

```
this_ := {}
// Many lines later
asize(this_, reccount())
```

Assignments to multiple dimensions can also be made more clear. In the following example, **iTest_** ends up as a two-dimensional array. The first dimension is controlled by the AADD() functions, and each call adds an array of three numbers which form the second dimension.

```
iTest_ := {}
aadd(iTest_, {1,2,3})
aadd(iTest_, {4,5,6})
aadd(iTest_, {7,8,9})
```

Whether you find the above technique clearer than an equivalent example, below, is a matter of personal preference.

```
iTest_ := {{1,2,3}, {4,5,6}, {7,8,9}}
```

We prefer the "all in one statement" technique when the list of elements is reasonably short and obvious in structure. The AADD() technique is better in situations where the list is too long to fit in one or two source code lines or when the structure is too convoluted to construct correctly.

An interesting side effect of the ability to create literal arrays is that the following syntax is acceptable.

```
i := 2
? {"one", "two", "three"}[i]  //  "two"
```

The curly braces delimit an array, the square braces delimit an element reference. Put them together and out pops a value.

Variable multi-dimensional structure

This section gets complicated, fast. You don't need to use or even understand variable structures to make good use of Clipper arrays. Feel free to skip this section and come back later when you feel you've mastered the intricacies of less ambitious arrays.

Using a database in the previous section to illustrate multiple dimensions obscures an important point. The dimensions of an array do not have to be uniform, consistent or symmetrical. You can make a huge mess anytime you want. However, the completely arbitrary nature of array structures allows you to do things in an efficient and elegant way that are ordinarily not even practical, much less pretty.

For a real life example, let's implement an array of family members. A family has a single surname followed by one or more given names, depending on marital status and the number of children.

Let's start by describing our motivation for wanting to use a multi-dimensional array with a variable structure. Suppose a Clipper software application needs to deal with "households" as a fundamental unit, for example, a membership system for an organization. The application needs to be able to produce standard reports based on wildly different family situations: Single individuals, married with no children, married with one or more children, single parent with one or more children.

It's preferable to pass a single array containing the complete family structure to functions in the application, rather than laboriously specifying a worst-case structure in each function call. The following are examples of calls to functions that could be written to handle the arbitrary family structure. In both cases the family information is stored in related DBF files. Here is the general structure to consider.

a FAMILY is a SURNAME
plus one or two SPOUSE-NAMES
plus zero or more CHILD-NAMES

In the first example, there's a call to LoadFamily(), which does the necessary database manipulations to create a single array that contains everything known about the family. This family array can then be passed to functions that know the structure and can act on it. For example, the purpose of the StdLetter() function could be to print a standard letter that's customized for different types of families.

```
/*
   Using a variable array structure
*/
cMember := AskMemNum()           //  Ask for member number
family_ := LoadFamily(cMember)   //  Load entire family
n := ChildCount(family_)         //  Count the kids
StdLetter(family_)               //  Send "standard" letter
```

The next example, implemented with multiple one-dimension arrays, requires a variable for each distinct component of the family structure. Sending this structure to the StdLetter() function is more complex, requiring three parameters. If we add more details to our family structure we will have to edit the source code, possibly significantly.

```
/*
   Without the use of a single family structure
*/
cMember := AskMemNum()            //  Ask for member number
```

```
surname := LoadSurname(cMember)  // Returns single string
spouse_ := LoadSpouses(cMember)  //  One or more names
kid_ := LoadKids(cMember)        //  Zero or more names
n := len(kid_)
StdLetter(surname, spouse_, kid_)
```

The **family_** array has the following variations.

```
{"Smith", {"John"}}             //  Single individual

{"Jones", {"Bob", "Sue"}}       //  Married, no kids

{"Davis", {"Jim"},;             //  Single parent,
   {"Mike", "Glen"}}            //  two children

{"Anders", {"Tina", "Steve"},;  //  Married,
   {"Cindy"}}                   //  one child
```

It isn't always a good idea to construct complex arrays in a single statement. The following are equivalent to the previous examples.

```
//  Single individual
family_ := {"Smith"}
aadd(family_, {"John"})

//  Married, no kids
family_ := {"Jones"}
aadd(family_, {"Bob","Sue"})

//  Single parent, two children
family_ := {"Davis"}
aadd(family_, {"Jim"})
aadd(family_, {"Mike","Glen"})

//  Married, one child
family_ := {"Anders"}
aadd(family_, {"Tina","Steve"})
aadd(family_, {"Cindy"})
```

The use of intermediate arrays can also help to clarify the creation of complex array structures.

```
surname := "Beck"
spouse := {"Gene","Jean"}
kids_ := {"Jake","Rick","Roxanne","Rita"}
family_ := {surname, spouse_, kids_}
```

The first element of the array is the family surname. The second element is an array of one or two names. One name implies a single individual, two names implies a married couple. The third and optional element is an array of names of children. Note that even when there is only a single spouse or child the name is still part of an array. A list of one is still a list. Given this structure definition we can make the following observations:

```
if len(family_) = 2          //  No children
if len(family_) = 3          //  One or more children
if len(family_[2]) = 1       //  Single individual

? family_[1]             //  Family surname
? len(family_[3])        //  Number of children in family
? family_[2,1]           //  Name of first parent or individual
? family_[2,2]           //  Name of other parent, if any
? family_[3,1]           //  Name of first child, if any
? family_[3,n]           //  Name of N'th child, if any
```

A function that displays a complete description of any given family structure looks like Listing 9.1. A series of preprocessor #define directives makes it much easier to deal with arrays.

Listing 9.1 Processing a "family" array

```
#define SURNAME  1
#define PARENT   2
#define CHILD    3
#define SPOUSE1  2,1
```

```
#define SPOUSE2  2,2

function ListFamily(f_)
local i
if len(f_[PARENT]) == 1
  ? f_[SPOUSE1] +" " +f_[SURNAME]
else
  ? f_[SURNAME] +": "
  ?? f_[SPOUSE1] +" and " +f_[SPOUSE2]
endif
if len(f_) > 2
  ? "Children:"
  for i := 1 to len(f_[CHILD])
    ?? " " +f_[CHILD, i]
  next i
endif
return nil
```

Given the family structures, listed previously, calls to ListFamily(family_) will display the following:

```
John Smith

Jones: Bob and Sue

Jim Davis
Children: Mike Glen

Anders: Steve and Tina
Children: Cindy
```

In the example we described an array containing only first and last names. This is potentially useful but not terribly so. The true power will be more apparent when we add another dimension — information specific to each name. Suppose that we also wish to keep track of a mailing address for each family, membership number and date of birth for each family member, plus occupation and employer for parents and school for each child. We won't go into details (that would make a whole chapter by itself), but here are some sample array definitions that will pique your interest if you're following the concept of variable structures. We've omitted the membership numbers and dates of birth for clarity. In practice these things can get as complex as you need to model the real world.

```
{{"Smith", "1234 Main Street", "Anytown"}, ;
   {{"Bob",   37, "ACME Corp", "Sales Manager"}, ;
    {"Sally", 36, "Ajax Systems", "Analyst"}}, ;
      {{"Mark", 14, "Park Jr High"}, ;
       {"Tom",   8, "South Elementary"}, ;
       {"Mary",  2}}}
```

Translation: Bob and Sally Smith, and their children Mark (age 14), Tom (age 8), and Mary (age 2), live at 1234 Main Street in Anytown. Bob is 37 and the sales manager at ACME Corp. Sally is 36 and an analyst at Ajax Systems. Mark attends Park Jr. High and Tom attends South Elementary. Mary does not go to school.

The incredible flexibility is apparent when you consider that the same array structure will hold Barney Taft, a 47-year old unemployed person with no known address, as well as the Parker family, with seven children and two careers. Poor Barney's array is listed below. We'll leave the fairly lengthy Parker array to your imagination.

```
{{"Taft"}, {"Barney", 47}}
```

Such complex array structures may hold "real world" data logically and efficiently, but what about accessing all that data? Is it equally complex? Not really. The trick is to eliminate most of the array from consideration and concentrate on the small segment that interests you at the moment. Trying to ingest the entire array at once is extremely difficult to do. Let's decompose the complex family array by establishing #define directives for the first dimension, as we did in the previous ListFamily() function.

```
#define SURNAME 1
#define PARENT  2
#define CHILD   3
```

So, the huge array is really just three nested arrays.

```
family_[SURNAME]   //  Array of address info
family_[PARENT]    //  Array of parent arrays
family_[CHILD]     //  Array of children arrays
```

Let's take the PARENT array and decompose it in the same manner.

```
#define SPOUSE1  2,1
#define SPOUSE2  2,2
```

This makes the somewhat complicated parent array easier to deal with — it's just a pair of arrays. PARENT is the second array inside the main family array, and SPOUSE1 is the first array within the PARENT array. Becoming more clear? Let's continue with the inner workings of the SPOUSE1 array.

```
#define SPOUSE1_NAME  2,1,1
#define SPOUSE1_AGE   2,1,2
#define SPOUSE1_EMP   2,1,3
#define SPOUSE1_JOB   2,1,4
```

This breaks the SPOUSE1, or "first spouse," array into four elements. There are no more arrays, so we are done. Since undefined array elements have the value NIL, we can quickly answer many questions.

```
if family_[SPOUSE2] = NIL      //  Single individual
if family_[SPOUSE1_EMP] = NIL  //  Spouse has no employer
```

Using #define makes the array references more clear. For example, the above looks like this after preprocessing. Not nearly so obvious.

```
if family_[2,1] = NIL
if family_[2,1,3] = NIL
```

We used single #define directives to represent repeating portions of the PARENT array, which is okay for one or two parents but not such a great idea for zero to "n" children. Here's the breakdown for the CHILD array, using a slightly different #define tactic.

```
#define    CHILD   3     // Child is 3rd element in main family
                         // array.
#define    NAME    1     // Child's name is first element
#define    AGE     2     // in a child array, age is second,
#define    SCHOOL  3     // name of school is third.
```

This allows us to refer to a child by number, rather than having to define a set of #define directives for each possibility.

```
? family_[CHILD, 6, NAME]     //  Name of sixth child
? family_[CHILD, 2, SCHOOL]   //  Second child's school
if family_[CHILD] = NIL       //  No children
? len(family_[CHILD])         //  Number of children
```

One more blow-your-mind example and we'll leave this fascinating concept alone. Suppose we want to keep track of *all* the schools each child attends? We can add another array inside of each CHILD array containing a list of zero to "n" schools attended! There is no practical limit to the amount of data that can be tucked away into a multi-dimensional array with a variable structure.

We haven't even touched on the even more powerful technique of storing record type indicators in the array, which allows us to store different structures within the same general structure. An example is storing completely different kinds of data for members versus non-members: Members have membership details while non-members have prospect information. The same outer structure (name, address, phone and so on) is used for everyone, but the inner structure is wildly different based on the value of the "membership flag" element. If yes, the inner structure contains membership number, date joined, dues paid and so on. If no, the inner structure contains date last contacted, who supplied the lead, etc. Some members may be officers in the organization, or inactive members, or trial members, so a "membership type" element indicates which kind of inner structure to expect. Yikes. Better stop conceptualizing and get back to the basics.

Storage considerations

In Summer '87 the memory storage mechanisms of arrays are fairly simple to understand. You always have to declare a variable in advance as an array, and specify its maximum size at the same time. Since there is no such thing as a literal array (where an array can be established and filled with values independent of being assigned to a memory variable), all references to arrays and array elements can be traced back to a particular array name. The programmer always explicitly creates array names and declares sizes. The ACOPY() function merely copies values from one array to another. Other functions that fill arrays with values (like ADIR() for directory information) need an additional initialization step where the arrays are created and sized prior to being used.

When an array is passed as a parameter, it isn't hard to see that it is working on the same position in memory, but is given a different name.

```
function Main
local one_[100]
MyFunc(one_)
return nil
```

```
function MyFunc(two_)
afill(two_, "")
return nil
```

In the Main() and MyFunc() examples above, **one_** and **two_** refer to the same position in memory. In Summer '87 the concepts of *equality* and *equivalence* are usually not a big deal because the programmer is totally responsible for creating and sizing the arrays. With respect to arrays these terms have the following meanings.

Equality: All the elements in one array are equal in value to all the elements in another array.

Equivalent: The two arrays occupy the exact same place in memory, meaning the two variable names refer to the same array.

one_ is equivalent to **two_** because they are different names for the same structure in memory. Consequently, equivalent arrays are "equal" only in a degenerate sense.

```
local one_[100], two_[100]
afill(one_, 0)
acopy(one_, two_)
```

In this example, two distinct arrays are created so there are two distinct structures in memory. After the ACOPY() function is finished we can say that the two arrays are equal but not equivalent. Arrays are equal only when each element has the same value in both. You cannot compare two arrays directly with =, >, <, and so on. You must compare each element individually.

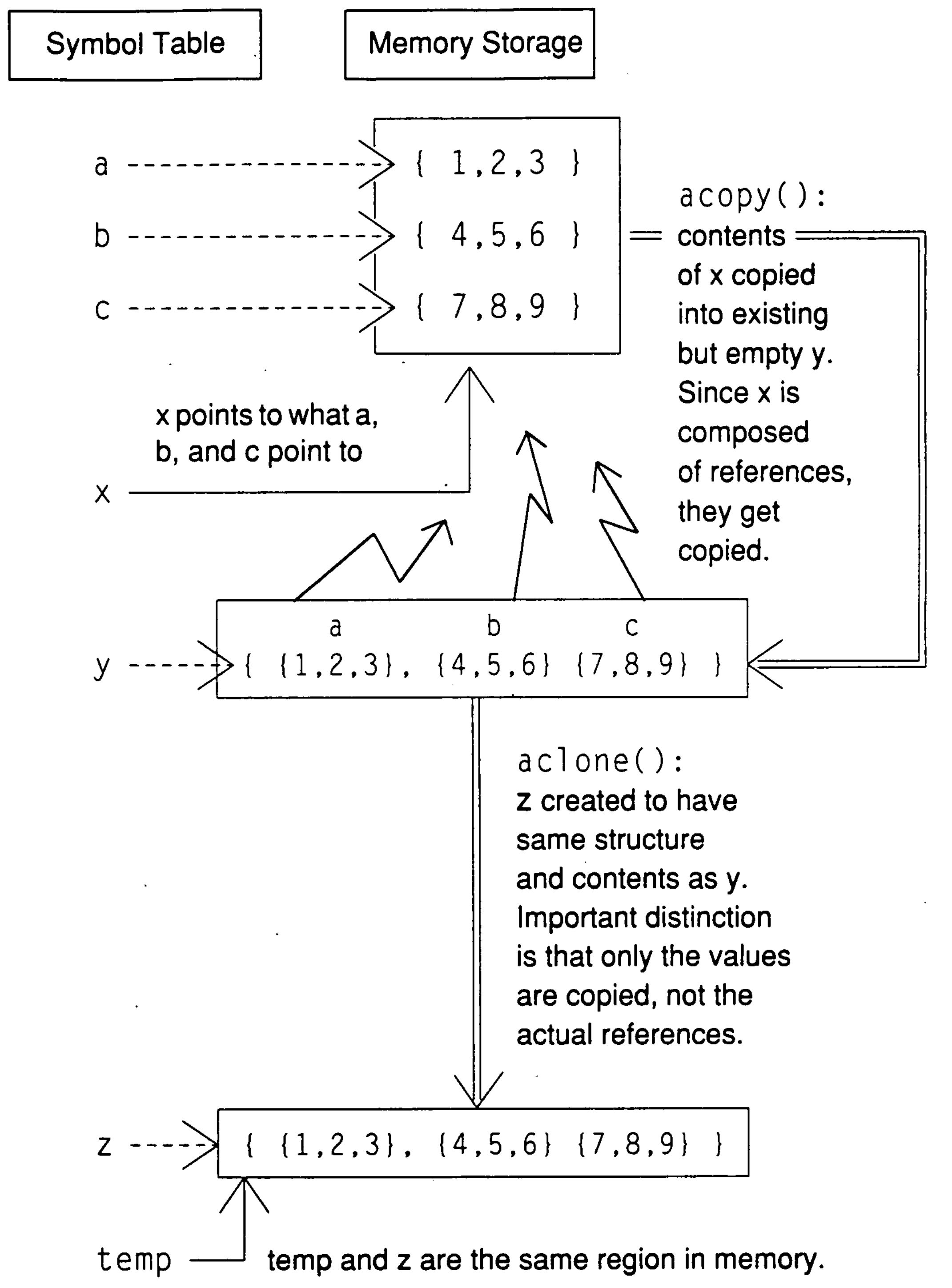
In Clipper 5 the previously unambiguous attributes of equality and equivalence are no longer so clear. Since arrays can contain other arrays, an array can be equivalent in some respects and equal in others. Consider the following.

```
// Create and fill three arrays,
// each containing three numbers.
a := {1,2,3}
b := {4,5,6}
c := {7,8,9}

// Create an array filled with existing arrays.
// Array x has three elements, each element is an
// array of three numbers.
x := {a, b, c}

// Create an empty array with same dimensions as the x array.
y := array(3, 3)

// Copy contents of array x into array y.
acopy(x, y)

// Create new array z with same dimensions as array y,
// fill with values found in array y.
z := aclone(y)

// Assign temp as an additional name for array z.
temp := z
```

After these array assignments have been made we can depict the resulting symbol table and memory contents (see Figure 9.3). Note that this is simplified and idealized; the actual symbol table and variable storage is considerably more complicated. If the variables being considered are **local** or **static** in scope, the symbol table becomes a non-issue.

Figure 9.3. Approximation of the symbol table and arrays in memory.

Dashed lines	-------	Direct assignment to a region in memory.
Single lines	———	A reference established to an existing region.
Double lines	═══	A process that uses array references.


Symbol Table
Memory Storage
a
b
c
{ 1,2,3 }
{ 4,5,6 }
{ 7,8,9 }
acopy():
contents of x copied into existing but empty y. Since x is composed of references, they get copied.
x points to what a, b, and c point to
x
y
a b c
{ {1,2,3}, {4,5,6} {7,8,9} }
aclone():
z created to have same structure and contents as y. Important distinction is that only the values are copied, not the actual references.
z
{ {1,2,3}, {4,5,6} {7,8,9} }
temp
temp and z are the same region in memory.

In this example there are seven array names but only six distinct array structures in memory. **a, b,** and **c** are simple arrays with one dimension. **x** is an array that contains the **a,** b and **c** arrays as elements. **x[1]** is an array that is equivalent to **a,** because **x** contains references to the other arrays. The ACOPY() function fills the **y** array with the values found in **x**, but since **y** was created explicitly with the ARRAY() function to be a three-by-three array, **x** and **y** are not equivalent (since they occupy two distinct places in memory) but are considered equal since they contain equal values in all elements. Important note: **y** contains references to **a, b,** and **c** just as **x** does, since ACOPY() copies references to subarrays and not values. Despite the fact that **x** and **y** are distinct arrays, only the references to **a, b,** and **c** are stored in two different places. **x[1]** and **y[1]** still both refer to array **a**. In contrast, **z** is a completely new copy of **x**, with no references to **a, b,** or **c**. As far as the values of elements are concerned, **z** is equal to both **x** and **y**. As far as references to places in memory, **z** is not equivalent to **x** or **y** or the **a, b,** and **c** arrays. Finally, **temp** is assigned **z** in its entirety, so **temp** and **z** are equivalent, referring to the same place in memory.

The only operator that can be used directly with array references is the double equal (==), which in this context is taken to mean "are equivalent." The regular equal operator (=) and all the variations such as greater-than, less-than, and so on are not valid for entire arrays.

Given the previous list of array assignments we can make the following observations.

```
temp == z        //  True, same place in memory.

x == y           //  False, distinct structures in memory.

x[1] == y[1]     //  True, same subarray references.

z == x           //  False, different subarray references.

z[1] == x[1]     //  True, values are equal even though
                 //  the arrays are not equivalent.
```

Let's examine one more situation: Passing arrays to functions. Assume the arrays from the previous example are still available.

```
function Testing
/*
   Assume all the array assignments to a,b,c,x,y,z,
   and temp are still around, as listed previously.
   To summarize:

     Array a contains {1,2,3}
     Array b contains {4,5,6}
     Array c contains {7,8,9}

     Array x contains {a,b,c}
     which are references to arrays a, b and c.
*/

?? a[1]                     // 1
?? x[1,1]                   // also 1
MyFunc(a, 0)
?? a[1]                     // now 0
?? x[1,1]                   // also now 0

MyFunc(aclone(b), 999)
?? b[1]                     // still 4

return nil

function MyFunc(name, value)
afill(name, value)
return nil
```

After the call to MyFunc() the **a** array contains zeros, since **a** is passed by reference. And since **x** and **y** refer to the same position in memory as **a** does, they refer to those zeros as well. **z** was created via ACLONE() and consequently was not affected by MyFunc(). In the next call to MyFunc(), the result of ACLONE(**b**) was passed rather than **b** directly, so no values were altered outside of MyFunc(). An unnamed clone of **b** was created for the call, was altered while in MyFunc(), and was released after the call. The return value of ACLONE() was passed directly as a parameter, so as far as the rest of Testing() is concerned it never existed.

Having a firm understanding of the difference between equivalence and equality will help prevent confusion when dealing with a complex system of arrays. This is especially true when passing arrays as parameters to functions. Unless you use ACLONE(), the functions are working with equivalent arrays, referring to the same position in memory.

This brings us to one more potentially confusing aspect of array references. Based on what you know about arrays, examine the following code fragment and determine what will happen.

```
function Main()
x := {10, 20, 30}
Foobar(x)
? x[2]               //  ?
return nil

function Foobar(y)
y := {"A", "B", "C"}
return nil
```

So far we've made array references sound like simple pointers. In fact, a reference to an array name is actually a reference to yet another reference and not directly to the values in the array. In the following example we've condensed the previous call to Foobar() into the essentials we need for this discussion.

```
x := {10, 20, 30} // As established in Main()
y := x            // Upon being accepted as a parameter in Foobar()
```

The **x** and the **y** are unique names, so a reference to **x** is, at the name level, distinct from a reference to **y**. A reference to **x** points us to a particular array in memory. A reference to **y** points to the same location. We have two pointers that happen to be pointing to the same thing. If we change an element value in array **y** we'll see the same change when we refer to that element via array **x**, since the **x** and **y** pointers are pointing to the same array.

You can think of this as **x** saying "go look in box #A for the memory address of the array to which I'm pointing." **y** is saying the same thing: "See box #A." It's box #A that actually contains the address of the array, not **x** and **y**. **x** and **y** just point to box #A.

Let's return to the initial "what will happen" question. In the code fragment the Foobar() function accepted **x** as a parameter and called it **y**. The function then turns around and gives the variable name **y** an entirely new value! Instead of altering an element within the array (which we know from previous discussions would have been reflected in **x**), the function assigns **y** to a new literal array. What happens? Contrary to what you might initially think, **y**'s pointer is merely changed to the new array that Foobar() created.

Back up in function Main() the **x** array remains unchanged. Therefore, the value that gets displayed is 20, not "B". Why? Because **x** and **y** don't actually point to arrays. They started out as pointing to the same pointer, which in turn pointed to an array. **y**'s pointer got switched to a different pointer, while **x**'s pointer stayed the same.

Returning to the box analogy, **x** is still saying to look at box #A. **y**, however, is now saying "look in box #B". Box #B knows where to find the new array. Box #A still has the address of the original array. **y** did indeed get a new value, but the link to **x** (which was indirect in the first place) has been severed.

Functions using arrays as parameters

Many functions use arrays as parameters. We will discuss each one briefly. You can use arrays as parameters when writing your own functions. See a later section in this chapter for more details.

We need to take a step back for a moment and note that you can always pass individual array elements as parameters just like any other variable.

```
s := {"123", "456", "789"}
n := val( s[3] )            // 789
```

You can even @..SAY..GET an array element.

```
for i := 1 to len(answer_)
  @ row() +1, 4 say "?"  get answer_[i]
next i
```

This section is devoted to functions that accept entire arrays as parameters. Since arrays are always passed by reference (as opposed to value), the function being called can alter the contents of the array. We can divide these functions into two broad classes: Those that are called to alter the contents of the array, and those that are called for some other effect.

Most of the functions are discussed only briefly; see your Clipper documentation or Norton Guides for syntax details and complete technical run-downs on each function.

Functions that alter the array

Many of the built-in Clipper functions designed to deal with arrays are used to alter the size or the contents of the target array.

AADD() increases the size of an array by one element. It can optionally assign a value to the new element. The new element is added to the end of the array with a NIL value. The value AADD() adds may be an array, but AADD() will add a reference to that array and not a copy of it. AADD() dynamically alters the length of the array while the array insert function, AINS(), does not.

```
a := {1,2,3}
? len(a)        //  3
aadd(a,)
? len(a)        //  4
? a[4]          //  Nil
aadd(a,"Z")
? len(a)        //  5
? a[5]          //  "Z"
```

ACLONE() makes a complete copy of any array, including multi-dimensional arrays, and places it in a new place in memory. ACLONE() creates a new array, while the array copy function, ACOPY(), does not. When ACLONE() encounters a subarray it creates a new copy of the subarray.

```
a := {1,2,3}
b := {9,8,7}
? a == b          //  False, different place in memory
x := a
? x == a          //  True, same place in memory
y := aclone(a)
? y == a          //  False, ACLONE() creates a new array
```

ACOPY() makes a copy of an array. The target array must already exist. If the source array is larger than the target, the additional elements are ignored. If the source is smaller than the target, the target elements are left as they were. When ACOPY() encounters a subarray it leaves a reference to the subarray, as opposed to a copy as ACLONE() does. ACOPY() can start copying at any point in the target array and copy only a specified number of elements. ACOPY() can also start copying at a specified point in the target array.

```
a := {1,2,3}
b := {9,8,7}
acopy(a, b)   //  b now contains {1,2,3}

c := {1,2,3,4,5,6,7,8,9}
d := {0,0,0,0,0,0,0,0,0}
acopy(c, d, 5, 2)         //  d now contains
                          //  {5,6,0,0,0,0,0,0,0}

acopy(c, d, 1, 3, 7)      //  d now contains
                          //  {5,6,0,0,0,0,1,2,3}
```

ADEL() deletes an element from an array. The size of the array is not affected. Elements below the deleted element are shifted up. The new last element has a value of NIL. If the deleted element is a subarray the entire subarray is deleted. The array sizing function, ASIZE(), is used to actually change the number of elements in the array.

```
a := {1,2,3,4}
? len(a)         //  4
adel(a, 2)       //  a now contains {1,3,4,nil}
? len(a)         //  4
? a[2]           //  3
? a[4]           //  NIL
```

ADIR() places directory information in separate arrays. The arrays must already exist. ADIR() will not increase or decrease the sizes of the target arrays. ADIR() also returns a count of the number of files in the directory that match a file specification. The count can be used to determine the optimum sizes of the target arrays. The general purpose directory loading function, DIRECTORY(), is a superior alternative. ADIR() is marked as a Summer '87 compatibility function and should be avoided.

```
file_ := {}
cnt := adir("*.DBF")  // Returns count of files matching *.DBF
asize(file_, cnt)     // Make array exactly large enough
```

```
adir("*.DBF", file_)
? file_[1]              // Name of first DBF file
```

AFIELDS() places database structure information in separate arrays. Similar to ADIR(), AFIELDS() does not alter the size of the arrays. The general purpose database structure loading function, DBSTRUCT(), is a better alternative. AFIELDS() is marked as a Summer '87 compatibility function and should be avoided.

```
use TOYS new
field_ := {}
type_ := {}
cnt := TOYS->(fcount())
asize(field_, cnt)
asize(type_, cnt)
afields(field_, type_)
? field_[1], type_[1]   //  Name and type of first field
```

AFILL() fills an array with a specified value. It does not alter the length of the array. You may specify the starting element and number of elements to process.

```
a := {1,2,3,4,5}
afill(a, 0)          //  a now contains {0,0,0,0,0}
afill(a, 9, 2, 3)    //  a now contains {0,9,9,9,0}
```

AFILL() operates on one dimension at a time, and will overwrite subarrays. Great care must be taken with multiple dimensions. The function in Listing 9.2 will safely assign the same value to every element in an array, including subarrays. It makes a recursive call to itself so there is a limit to the number of nested subarrays that can be processed.

Listing 9.2. An AFILL() for arrays with multiple dimensions

```
function M_afill(a_, value)
/*
  Multi-dimension afill()
*/
```

```
local i
for i := 1 to len(a_)
    if valtype(a_[i]) == "A"
      M_afill(a_[i], value)
    else
      a_[i] := value
    endif
next i
return nil
```

AINS() inserts an element into an array. It does not alter the size of the array. The new element has a value of NIL. Elements beneath the one inserted are shifted down and the last element value is lost. The array sizing function, ASIZE(), is used to actually change the number of elements in the array.

```
a := {1,2,3,4}
? len(a)          //  4
ains(a,3)         //  a now contains {1,2,nil,3}
? len(a)          //  4
```

ARRAY() creates an array of specified dimensions. The array is filled with NIL values for all elements.

```
a := array(3)     //  a now contains {nil, nil, nil}
b := array(3,2)   //  b now contains
                  //    {{nil,nil}, {nil,nil}, {nil,nil}}
```

ASIZE() adds or removes elements in an array. If an array is made larger, the new values will be NIL. If an array is made smaller the values will be lost.

```
a := {}
? len(a)         //  0
asize(a, 100)
? len(a)         //  100
asize(a, 5)
? len(a)         //  5
```

ASORT() sorts the contents of an array. The elements can be of any data type but must be all of the same type to sort properly. By default, ASORT() sorts only the first dimension of a multi-dimension array. You may specify the starting element and number of elements to process.

```
a := {9,8,7,1,2,3,6,5,4}
asort(a)
//  Order is now {1,2,3,4,5,6,7,8,9}

b := {8,2,1,9,4,3,7,6,5}
asort(b, 4, 3)
//  Order is now {8,2,1,3,4,9,7,6,5}
//
```

You may specify a code block which determines the sort order. The code block can be used to change the sort order from the default of ascending, or to base the sort decisions on the contents of subarrays. The code block is given two values to compare. The code block should return true if the values are considered to be in order, or false if not. For example, the following code block can be used to sort in descending order. The block will return true if the first value is greater than the second value. Written this way, the bDescend code block will sort numbers, character strings, dates and even logical values.

```
bDescend := { |a,b| a > b }
```

The code block is sent to ASORT() as the fourth parameter. If you do not want to specify the second and third parameters (the starting element and count) you must still include the commas that separate the parameters.

```
a := {9,8,7,1,2,3,6,5,4}
asort(a,,, bDescend)
//  Order is now {9,8,7,6,5,4,3,2,1}
```

The code block can get pretty fancy, doing anything it needs to determine if the two values are in sorted order, as long as it ultimately returns true or false. Here's a code block that will sort an array of full names by last name.

```
bLastName := { |a,b| LastName(a) < LastName(b) }
```

Hey — that's cheating! Not really. There's no rule that says a code block has to do everything inside the block. Here is the source code for the LastName() function.

```
function LastName(name)
return substr(name, rat(" ", name) +1)
```

Armed with a fancy code block that handles last names we can sort an array of names.

```
a := {"Mr. Alex Smith", ;
      "Heather Sue Anderson", ;
      "Tracy J. Doe"}

asort(a,,, bLastName)

//  Order is now {"Heather Sue Anderson", ;
//                "Tracy J. Doe", ;
//                "Mr. Alex Smith"}
```

Sorting within multidimensional arrays

It is possible to use a code block to help overcome ASORT()'s inability to sort on anything but the first dimension of an array. Let's take the oft-used example of ranking the members of a bowling league. After reading this chapter on arrays, a crafty programmer decided to store his team's names and scores in a multidimensional array, like so.

```
team := {}
aadd(team, {"Earl", 224, 256, 198, 202})
aadd(team, {"Bud", 290, 278, 210})
aadd(team, {"Dick", 178, 201, 199, 207, 200})
```

Each team member occupies an element in the team array. Each member element is a subarray containing the member's name and a list of scores. Pretty elegant, thought the programmer, there's no wasted space because the structure neatly accommodates the fact that not everyone plays the same number of games. But how do you sort the team by name? ASORT() code blocks come to the rescue.

```
bSort := { |a,b| a[1] < b[1] }
asort(team,,, bSort)
```

ASORT() will pass this block two elements. In the case of the team array, that's two arrays. Instead of trying to compare the two entire arrays the block will compare the first element of each array — the name. ASORT() will sort the arrays within the team array by comparing the names of team members.

Here's a challenge: Using ASORT() rank the team array by most recent score, sorting them from the highest to lowest. Remember that not all members have played the same number of games. You can assume, however, that each member has played at least one game.

Functions that don't alter the array

Other built-in Clipper functions use arrays as parameters but don't alter the contents in any way.

ACHOICE() displays array contents in a scrolling window. A second, optional array specifies which elements in the first array may be selected. ACHOICE() returns the number of the element that was selected.

```
choices := {"Add", "Change", "Delete", "Quit"}
n := achoice(1,1,4,8, choices)
```

Chapter 13, "The Art of the User Interface," goes into more detail about user interface programming.

ASCAN() searches the contents of an array. You may specify the starting element and number of elements to process. ASCAN() will stop at the first element that matches the scan value. ASCAN() returns the element number or zero if no elements match.

```
//    1 2 3 4 5 6 7 8 9
a := {1,1,1,1,2,1,1,2,1}
? ascan(a, 2)              // 5
? ascan(a, 2, 6)           // 8
? ascan(a, 2, 1, 4)        // 0
```

Instead of a value for which to scan you may specify a code block. ASCAN() will pass to the code block each element of the array, one at a time. If the code block returns true, the scan will stop. If false, the scan continues. Here's an example that scans an array for the first element value that is even.

```
//    1 2 3 4 5 6 7 8 9
a := {7,9,3,5,6,3,4,7,1}
n := ascan(a, { |n| n%2 == 0 } )
? n     //  5
? a[n]  //  6
```

ATAIL() returns the value of the last element in the array. This is a handy alternative to using the LEN() function to return the last element number. The following examples are equivalent.

```
chars_ := {"A", "B", "C"}
? chars_[len(chars_)]       //  "C"
? atail(chars_)             //  "C"
```

Note that you are able to eliminate the need for an extra reference to the array. This allows for cleaner syntax and code that's easier to understand.

DBCREATE() creates an empty database structure based on the contents of an array. The array is two dimensional. The "inner" dimension is an array of field specifications: name, type, width and decimals. The "outer" dimension is an array of these arrays, one for each field. The following creates an INVENT.DBF file containing five fields.

```
stru := {}
aadd(stru, {"id",      "C",  6, 0})
aadd(stru, {"descr",   "C", 30, 0})
aadd(stru, {"price",   "N", 10, 2})
aadd(stru, {"updated", "D",  8, 0})
aadd(stru, {"active",  "L",  1, 0})
dbcreate("INVENT", stru)

? len(stru)     // 5
use INVENT new
? fcount()      // 5
```

See the section titled **Arrays and Databases** in this chapter for a more detailed discussion of DBCREATE() and DBSTRUCT().

DBEDIT() displays records in a window, format based on a large collection of arrays. The Nantucket manual states that DBEDIT() is a compatibility function and therefore, is no longer recommended as a programmable browse facility. The TBrowse object class is a vastly superior alternative, so don't waste any time with DBEDIT(). See Chapter 28, "Obsolete Commands and Functions," for a complete list.

LEN() determines the length (number of elements) of an array. LEN() returns the length of an individual element if you include an element reference.

```
a := {"A","B","CDEFG", "Z"}
? len(a)                    //  4
? len(a[3])                 //  5
```

TYPE() determines the type of an expression, and will return "A" if it's an array. Using TYPE() on an individual array element returns the type of the element. The VALTYPE() function is preferable to TYPE() and should be used instead. See Chapter 4, "Data Types," for more details. TYPE() can operate only on private and public variables. It cannot operate on local and static variables.

```
a := {1,2,3}
? type("a")     //  "A" for array
? type("a[1]")  //  "N" for number
```

VALTYPE() determines the type of data returned by an expression, will return "A" if an array. As with TYPE(), VALTYPE() will work with the entire array as well as an individual element. VALTYPE() is preferable to TYPE() because it can operate on local, static, private and public variables.

```
a := {1,2,3, {"A","B","C"}}
? valtype(a)                 //  "A" for array
? valtype(a[4])              //  "A" for array
? valtype(a[4,1])            //  "C" for character
```

Functions that return arrays

Some functions use arrays to hold return values. We will discuss each one briefly. You can return arrays when writing user-defined functions. See a later section in this chapter for writing user-defined functions that use arrays. Three Clipper functions actually create and fill a new array. Several others don't create new arrays but return a reference to the array.

ACLONE() returns a complete copy of an array, including subarrays. ACLONE() is distinct from ACOPY(), which copies only into an existing array. See previous discussion about array storage considerations in this chapter.

```
a := {1,2,3}
b := aclone(a)
```

DBSTRUCT() creates an array containing database structure information. It is th opposite of the DBCREATE() function. DBSTRUCT() returns a two-dimensiona array, one array containing field specifications (name, type, width, decimals) fo each field in the database. If no database is selected the function returns an empty array. A preprocessor #include file, DBSTRUCT.CH, is supplied to make it easier to program with the database structure array.

```
#include "DBSTRUCT.CH"
use VENDOR new
stru := dbstruct()
? stru[1, DBS_NAME]          //  Name of first field
? stru[1, DBS_TYPE]          //  Type of first field
? stru[fcount(), DBS_LEN]    //  Length of last field
```

DBCREATE() can be used with DBSTRUCT() to easily make a temporary copy of a database structure.

```
dbcreate("TEMP", VENDOR->(dbstruct()))
```

See the section titled **Arrays and Databases** in this chapter for a more detailed discussion of DBCREATE() and DBSTRUCT().

DIRECTORY() creates an array containing directory information, based on an optional file specification. The array it returns is structured similar to DBSTRUCT(). The array has two dimensions, the inner array contains name, size, date, time and attributes, and the outer array contains an inner array for each file in the directory. If no files match the specification this function returns an empty array. A preprocessor #include file, DIRECTRY.CH, is supplied to make it easier to program with the directory array. This chapter contains some comprehensive examples of the DIRECTORY() function. See also Chapter 22, “Disks and Directories” for a complete discussion.

```
#include "DIRECTRY.CH"
dir := directory("*.NTX")

? dir[1, F_NAME]          // Name of first index
? dir[1, F_SIZE]          // Size in bytes
? dir[1, F_DATE]          // Date stamp
? dir[1, F_TIME]          // Time stamp
? dir[1, F_ATTR]          // File attributes

? dir[len(dir), F_NAME]   // Name of last index
```

ACOPY(), ADEL(), AEVAL(), AFILL(), AINS(), ARRAY(), ASIZE(), and ASORT() all return a reference to the array being processed. This is occasionally useful and better than returning no value at all. In the following example, an array called **q** is filled with empty strings ("") and a reference to **q** is passed to the LEN() function.

```
for i := 1 to len(afill(q, ""))
  //
next i
```

Some programmers will find this feature irresistible, probably due to C language influence. All it does is allow you to both fill and find the length of the array and then use the length to bound a for..next loop. Obfuscation or elegance?

User-defined functions and arrays

Nantucket's sample functions

Nantucket supplies a collection of user-defined functions that deal with arrays. The \CLIPPER5\SOURCE\SAMPLE subdirectory contains ARRAY.PRG which includes the following functions. (This list may change over time with new releases of Clipper 5.)

ABROWSE()	Uses TBrowse to view contents of an array.
ABROWSEBLOCK()	Used in ABROWSE().
ABLOCK()	Used in ABROWSE().
ASKIPTEST()	Used in ABROWSE().
AMAX()	Subscript of element with highest value.
AMIN()	Subscript of element with lowest value.
ACOMP()	Compares all elements in array to a value.
DIMENSIONS()	List of dimensions in an array.

ABROWSE() and ABLOCK() are discussed in detail in Chapter 25 "The TBrowse Object Class". You may find the other functions in ARRAY.PRG useful as-is or as examples of general purpose user-defined functions.

Creating your own functions

You can make use of any array concepts you learn in your own user-defined functions. Your functions can accept arrays as parameters, and your functions can return arrays back to the calling routine. This opens a wide array (ahem) of possibilities for improved code efficiency and elegance.

Listing 9.3 contains an example of a user-defined function that accepts an array of strings and displays them in a box just large enough to hold them.

Listing 9.3 A function that uses an array to display messages

```
function Pugilist(r, c, msg_)
/*
   Display list of message lines in a box.
   Start display at screen coordinates r,c.
   Function will center message if r or c not specified.
   Array msg_ contains message lines.
*/
local i, width := 0

/*
   Find length of longest line
*/
for i := 1 to len(msg_)
```

```
  width := max(width, len(msg_[i]))
next i

/*
   If row or column not specified,
   calculate starting row and column
   that will center message on screen
   (maxRow() and maxCol() are supplied by the system).
*/
if r == nil
  r := (maxRow() -len(msg_)) /2
endif
if c == nil
  c := (maxCol() -width) /2
endif

/*
  Draw box large enough to contain message,
  then display each message line.
*/
@ r -1, c -1 to r +len(msg_), c +width
for i := 1 to len(msg_)
  @ r +(i -1), c say msg_[i]
next i
return nil
```

A typical call to the function looks like this.

```
Pugilist(5, 20, {"That customer number",  ;
                 "is not defined. Enter", ;
                 "another or press ESC",  ;
                 "to return to the menu."})
```

The resulting box will be just wide enough to accommodate the longest line, and will contain the four lines specified in the array. To take advantage of the function's ability to center the box in either direction you simply skip the parameter (but keep the comma!)

```
Pugilist(,, {"This","will be","in center of screen."})
Pugilist(, 60, {"Centered vertically","in column 60."})
Pugilist(7,, {"Centered","horizontally in row 7."})
```

See Chapter 13, "The Art of the User Interface," for more examples of functions that use arrays in this way.

Your user-defined functions may also return arrays. Here's an example of a function that returns an array of field values that match a given key. You pass the field value you are interested in and the field and key value the index is based upon. The function, shown in Listing 9.4, is designed to answer the request, "Give me all the parameter-1 where parameter-2 is equal to parameter-3."

Listing 9.4 A function that returns an array

```
function Gimme(what, where, when)
/*
   Returns array filled with the "what" field,
   for all records where the "where" condition is true
   when compared to the value specified by "when".
*/
local these_ := {}
seek when
do while &where. = when
  aadd(these_, &what.)
  skip
enddo
return these_
```

Here's an example of Gimme() in practice. We're looking for a list of customer names where the state is Minnesota. We assume that the customer database has name and state fields and is indexed on state.

```
who_ := CUST->(Gimme("name", "state", "MN"))
```

The names of customers in Minnesota will be in the **who_** array. **who_** can be passed on to a function that displays arrays, perhaps based on ACHOICE() or an array-based TBrowse. Advantages: The rest of the routine can mess around with the simple **who_** array rather than a database and index, and the Gimme() function can process a wide variety of similar requests. Disadvantages: **who_** will grow to be as large as the total number of matching records, which can be more than available memory can handle, or more than the 4,096 element maximum for arrays.

The listing 9.4 version of the Gimme() function uses macros to perform the logical comparison and array assignment. This was done this way to keep the basic idea easier to understand. A code block is a much better alternative because it is considerably faster and more flexible than a macro. Here's the function again, in Listing 9.5, this time using code blocks.

Listing 9.5 Same function as before, without macros

```
function Gimme(bWhat, bWhere, cStart)
/*
   Returns array filled with the result of the
   "what" code block, for all records where the
   "where" code block returns true.
   Start processing by seeking the "start" value.
*/
local these_ := {}
seek cStart
do while eval(bWhere)
  aadd(these_, eval(bWhat))
  skip
enddo
return these_
```

Here's a new Gimme() called with code block logic.

```
who_ := CUST->(Gimme( {|| name }, ;          // What to return
                      {|| state == "MN"}, ;  // Condition
                      "MN" ))                // Where to start
```

The only obvious improvement is eliminating macros. The use of code blocks actually makes the function call look more complex. Code blocks, however, make it easier to allow flexibility in the logic and can operate using only local and static variables, something macros can't always do.

See the section called "Arrays and Databases" in this chapter for a complete discussion on this technique.

The AEVAL() function

Arrays may be processed by passing element values through a code block via the AEVAL() function. AEVAL() evaluates the specified code block for each element in an array. You may optionally specify the starting element and number of elements to process. AEVAL() does not return a value, just a reference to the array that was processed. See Chapter 8, "Code Blocks," for a complete discussion.

AEVAL() passes each array element, one at a time, to the code block. Here's an example that lists the contents of the array.

```
a := {1,2,3}
aeval(a, { |n| qout(n) } )
```

The QOUT() function is called three times, once for each element in the array. QOUT() simply displays whatever is passed to it, so the result is the contents of the array being listed to the screen. AEVAL() also sends a second parameter to the code block, the element number currently being processed. This is often very useful. Here's the previous example again, this time using the element number.

```
a := {"X","Y","Z"}
aeval(a, { |n, e| qout(e, n) } )

// Resulting display-

    1   X
    2   Y
    3   Z
```

Here's another simple example that sums the numbers in the array.

```
a := {1,2,3,4,5,6,7,8,9}
sum := 0
aeval(a, { |n| sum += n } )
? sum                          //  45
```

The starting element parameter allows us to start processing beyond the first element. Here we start the summing process at element number six.

```
a := {1,2,3,4,5,6,7,8,9}
sum := 0
aeval(a, { |n| sum += n }, 6)
? sum                          //  30
```

The count parameter allows us to process less than the entire array. Here we sum only the first four elements.

```
a := {1,2,3,4,5,6,7,8,9}
sum := 0
aeval(a, { |n| sum += n },, 4)
? sum                          //  10
```

The following is a simple but useful user-defined function that uses AEVAL() to determine the length of the longest character string in an array.

```
function Longest(a_)
/*
   Return length of longest string in array.
*/
local n := 0
aeval(a_, { |s| n := max(len(s), n) } )
return n
```

Here's an example of a call to Longest().

```
msg_ := {"A message", ;
        "made up of several", ;
         "lines of text."}

? "Length of longest element: "
?? Longest(msg_)                  //  18
```

The Clipper manual gives some interesting but potentially confusing examples of AEVAL(). Let's take a look at each example in more detail, starting with Listing 9.6.

Listing 9.6 Example as it appears in Clipper Reference Manual

```
#include "Directry.ch"
//
local aFiles := directory("*.dbf"), nTotal := 0
aeval(aFiles, ;
  { |aDbfFile| ;
  qout(padr(aDbfFile[F_NAME], 10), aDbfFile[F_SIZE]), ;
  nTotal += aDbfFile[F_SIZE] ;
  };
)
//
?
? "Total Bytes:", nTotal
```

From the top: The #include preprocessor directive has the compiler bring in some #define directives that help make the output from the DIRECTORY() function easier to deal with. The references to F_NAME and F_SIZE come from DIRECTRY.CH.

The next line assigns the output of the DIRECTORY() function to an array called **aFiles**. The DIRECTORY() function is asked for an array of files in the current directory that match "*.DBF". Also on this line another variable, **nTotal**, is initialized to zero. **nTotal** will be used within the code block to accumulate file sizes.

The array returned from DIRECTORY() is a two-dimensional array. The inner arrays are comprised of the file name, size, date, time and attributes. For example, if there were three files matching *.DBF, DIRECTORY() will return an array with three elements, one element for each matching file. Here's an example.

```
{ {"DATA.DBF", 12654, 07/19/90, "14:25:06", "A"}, ;
  {"TEST.DBF",  3654, 06/21/87, "01:15:45", "A"}, ;
  {"ABCD.DBF", 87234, 01/12/62, "11:02:17", "A"} }
```

Note that the dates will be actual date types and not character strings or numbers. Based on this structure, the #define directive for F_NAME refers to the first element, and F_SIZE to the second.

Armed with this knowledge of what DIRECTORY() is giving us we can tackle the call to AEVAL(). The **aFiles** array is passed to the AEVAL() function, which in turn passes each element in **aFiles** through the code block. While inside the code block, the array element is called **aDbfFile**. Remember that the elements inside **aFiles** are themselves arrays, so the code block is being passed a five-element array each time it is evaluated. The first statement in the code block is a QOUT() function. QOUT() is the function equivalent of the "?" command. Code blocks can't handle direct commands. If they could, the line may have been written like the following.

```
? padr(aDbfFile[F_NAME], 10), aDbfName[F_SIZE]
```

The PADR() function pads the right side of the filename with spaces, in this case to a length of ten. This line simply displays the name and size of the current file.

The last line in the code block takes the **nTotal** variable (initialized to zero in the local statement) and adds the size of the current file. If the sample DIRECTORY() array described previously is used with the example, the following will be displayed.

```
DATA.DBF   12654
TEST.DBF    3654
ABCD.DBF   87234

Total Bytes: 103542
```

The example can be written without the use of the AEVAL() function. The following code is functionally equivalent. This will help explain what is going on inside the code block. The FOR..NEXT loop replaces the AEVAL() function. AEVAL() also made the reference to the **aFiles** array somewhat easier to manage.

```
local i
for i := 1 to len(aFiles)
  ? padr(aFiles[i, F_NAME],10)
  ?? aFiles[i, F_SIZE]
  nTotal += aFiles[i, F_SIZE]
next i
```

The second example for AEVAL() in the Clipper manual is somewhat more simple than the directory listing routine we just discussed. This example (shown below) takes the same output from DIRECTORY() and trims it down to be a single array of filenames from the more complex two-dimensional array structure.

```
#include "Directry.ch"
//
local aFiles := directory("*.dbf"), aNames := {}
aeval(aFiles, ;
  { |file| aadd(aNames, file[F_NAME]) } ;
)
```

Once again we #include the DIRECTRY.CH file which supplies the F_NAME constant. The **aFiles** array is assigned the output from DIRECTORY(), and **aNames** is initialized to be an empty array. AEVAL() is then called upon to pass each element of **aFiles** through a code block which uses AADD() to add an element to the **aNames** array, and assign it the filename. From DIRECTRY.CH, F_NAME is #defined to have a value of 1, and element one of the file array contains the filename. Remember that each element in the array returned by DIRECTORY() is a five element array. The net result is the **aNames** array contains a list of filenames.

Multi-dimensional arrays with variable structure

Clipper's ability to manipulate "ragged" arrays, where asymmetrical arrays can be nested within other arrays, allows us to perform what appear to be amazing programming feats with very little effort. This portion of the chapter provides an example of a real world use for such arrays.

Arrays with multiple dimensions are fairly easy to understand when the dimensions are uniformly and consistently structured. A good example of a two-dimensional array is the lowly database file: The overall file can be thought of as an array of records, with a record being an array of field values. Arrays with a variable structure are much more complicated. As a familiar example, consider the MS-DOS tree-structured directory.

If we want to load into memory the complete directory structure of any arbitrary drive volume, we should probably start with the directory entries in the root directory. If we load them into a single dimension array called FILES_, then FILES_[2] is the name of the second entry in the root directory. However, some of the entries will be subdirectory names which force us to load yet another set of directory entries. So, some of the FILES_ elements will have a second dimension to hold yet another array of directory entries. Since subdirectories can also contain more subdirectories this process needs to be repeated as many times as necessary. This is where the variable structure comes into play. The following example shows an array called FILES_ being filled with a simple directory structure.

```
files_[1]       := "AUTOEXEC.BAT"
files_[2]       := "CONFIG.SYS"
files_[3,1]     := "DOS"
files_[3,2,1]   := "APPEND.COM"
files_[3,2,2]   := "ASSIGN.COM"
files_[4]       := "COMMAND.COM"
```

The first two elements are simple file names. The third is a two-dimensional array, one dimension for the subdirectory name and one for an array of file names within that subdirectory. The fourth element goes back to a regular filename. This kind of notation, while "traditional" in the Summer '87 sense, is difficult to grasp. The following code is functionally identical and does a better job of indicating the relationships among the elements.

```
files_[1] := "AUTOEXEC.BAT"
files_[2] := "CONFIG.SYS"
files_[3] := {"DOS", ;
                {"APPEND.COM", "ASSIGN.COM"} ;
             }
files_[4] := "COMMAND.COM"
```

We don't have to stop there. The following code is also functionally identical to the other examples.

```
files_ := {"AUTOEXEC.BAT", ;
           "CONFIG.SYS", ;
           {"DOS", ;
              {"APPEND.COM", "ASSIGN.COM"} ;
           }, ;
           "COMMAND.COM"}
```

We prefer the "all in one statement" technique when the list of elements is reasonably short and obvious in structure. Use of the array element adding function, AADD(), is better in situations where the list is too long to fit in one or two source code lines or when the structure is too convoluted to construct correctly. Here is the same directory structure array constructed using the AADD() function.

```
files_ := {}
aadd(files_, "AUTOEXEC.BAT")
aadd(files_, "CONFIG.SYS")
aadd(files_, {"DOS", {"APPEND.COM", "ASSIGN.COM"}})
aadd(files_, "COMMAND.COM")
```

What does another subdirectory level look like? The following example shows \DOS\UTILS containing DR.COM and ASK.COM.

```
files_ := {"AUTOEXEC.BAT", ;
           "CONFIG.SYS", ;
           {"DOS", ;
              {"APPEND.COM", ;
               "ASSIGN.COM", ;
                {"UTILS", ;
                  {"DR.COM", ;
                   "ASK.COM"} ;
                } ;
              } ;
           }, ;
           "COMMAND.COM"}
```

How do we reference elements in an array with a variable structure? You can't make a reference until you first check that you're not dealing with another array. If a reference returns a character string, you're done. If a reference returns an array, you need to dive a level deeper and start checking those array elements as well. The following is an example of a function, shown in Listing 9.7, that displays the contents of an array structured in the manner just described. To improve readability, each subdirectory gets indented three spaces. A bit further into this section we'll have a function create the necessary array automatically.

Listing 9.7 A function which lists an array with variable structure

```
function ListDir(dir_, level)
/*
   List the contents of an array containing a
   directory structure. This function uses a
   recursive call to itself.
*/
local i
if level == nil
  level := 0
endif
```

```
for i := 1 to len(dir_)
  ? space(level *3)
  if valtype(dir_[i]) == "A"
    ?? dir_[i, 1]
    ListDir(dir_[i, 2], level +1)
  else
    ?? dir_[i]
  endif
next i
return nil
```

The first time ListDir() is called it will not have a parameter, so we default the level of indentation to zero. A careful reading of the example shows a recursive call to the ListDir() function. ListDir() keeps calling itself until it runs out of array dimensions. The array was deliberately structured in such a way that any given level in the array is structured exactly like the level above it. This allows us to strip off the subdirectory name and pass the associated array of subdirectory entries along for further processing. This can go on indefinitely. The level parameter is increased by one each time ListDir() calls itself. Consequently, each subdirectory processed gets indented one level further than its parent (see Figure 9.4).

Figure 9.4 Typical output from the ListDir() function

```
CLIPPER5
   BIN
      UTILS
         LIB.EXE
         AD.EXE
      CLD.EXE
      CLIPPER.EXE
      RMAKE.EXE
      RTLINK.EXE
   INCLUDE
```

```
    ACHOICE.CH
    BOX.CH
    STD.CH
  LIB
    CLIPPER.LIB
    DBFNTX.LIB
```

So far we've been discussing how the array looks when filled and how to list its contents. But how does the subdirectory structure get loaded in the first place? The trick is in the extremely nifty DIRECTORY() function. The DIRECTORY() function is a major improvement over ADIR(). To use DIRECTORY() most effectively you should use the #include preprocessor directive to include the DIRECTRY.CH file, which contains handy constants for dealing with the arrays the function returns.

The DIRECTORY() function returns an array of arrays. Each element in the "outer" array stands for an entry in the directory. Each directory entry is described by an array of five values: name, size, date, time, and attributes. Here's an example of three files. (The dates are date-type values, not text strings or numbers.)

```
{ {"ONE.DAT",   12654, 03/09/90, "12:24:16", "A"}, ;
  {"TWO.DBF",    3654, 12/11/89, "01:25:40", "A"}, ;
  {"THREE.BAT",    24, 06/19/90, "07:12:23", "A"} }
```

Subdirectories have a "D" attribute, so we can detect them easily enough. However, each subdirectory also has those goofy "." and ".." entries, which stand for "me" and "my parent directory". These are not terribly useful in a directory listing so we can skip them.

Based on the definition of how to display the directory structure, we can write a function that creates such a structure in an array. We need to create an element for each file in the directory. Whenever we encounter a subdirectory we need to create an array instead of an element. The array contains the name of the subdirectory followed by an array of the file names within that subdirectory. Just to complicate matters further, the contents of that subdirectory may include more subdirectories! Sounds like recursion to me. See the source code in Listing 9.8.

Listing 9.8 A function that creates a variable sructure array

```
#include "DIRECTRY.CH"

function LoadDir(path)
/*
   Return array containing entire directory structure
   of drive volume. This function uses a recursive call
   to itself. You do not need to specify the path unless
   you want to load only a subset of the drive volume.

*/
local i, name, d_, r_ := {}

if path = nil
  path := "\"
endif

//  Load contents of the specified directory path,
//  including any subdirectory entries that might be there.
d_ := directory(path +"*.*", "D")

//  Loop once for each entry in directory.
for i := 1 to len(d_)
  name := d_[i, F_NAME]

   //  If the file attribute indicates this
   //  is a subdirectory entry, special handling is needed.
   if ("D" $ d_[i, F_ATTR])

   //  Skip the "." and ".." entries.
   //  (The $ operator means "contained in")
   if .not. (name $ "..")
```

```
      //  Add the subdirectory name to the array
      //  and call the directory loader function to
      //  return the array of file names.
       aadd(r_, {name, LoadDir(path +name +"\")})
      endif

  //  If the file isn't a subdirectory name,
  //  add it to the array of file names.
  else
    aadd(r_, name)
  endif
next i

return r_
```

The above code does a lot in a small number of lines. The recursion makes the whole thing possible, but at the expense of making it harder to understand. Keep in mind the general structure of the array that it builds. The repeating structure

```
{ FILE, FILE, {SUBDIR, {FILE, FILE, {SUBDIR, {...
```

represents a DOS directory structure like the following.

```
FILE
FILE
SUBDIR
  FILE
  FILE
  SUBDIR
    :
    etc
```

Multidimensional arrays with a variable structure are not trivial, but once mastered, the technique can be put to use in situations that are almost impossible to handle efficiently any other way.

Arrays and databases

Clipper 5 supplies us with the fundamental functions needed to create and manipulate databases via arrays, similar to the way the low-level file functions are implemented — just the basics, nothing fancy.

Outbound: DBCREATE(), creates an empty database file based on an array containing the structure. It writes out a database structure.

Inbound: DBSTRUCT(), returns an array containing the structure of a specified database file. It reads in a database structure.

These two functions are the inverse of one another. The following example demonstrates a new database called CLONED being created from the structure of ORIGINAL. The DBSTRUCT() function reads the structure of the currently selected database and returns an array containing that structure. The DBCREATE() function creates the specified database file with a structure based on the array.

```
//  An alternative to "copy structure to CLONED"
use ORIGINAL new
dbcreate("CLONED", dbstruct())
```

The array that these functions use has the following general structure. Each element in the array is another array, one for each field in the database. The LEN() of the array is the number of fields.

```
{field1_, field2_, .. fieldn_}
```

Each field array has the following structure.

```
{name, type, length, decimals}
```

So, an example of an array containing a database structure looks like the following. Multidimensional arrays are very handy for such things.

```
{ {"ID",    "C",  6, 0}, ;
  {"NAME",  "C", 30, 0}, ;
  {"QTY",   "N",  4, 0}, ;
  {"PRICE", "N",  8, 2}, ;
  {"DUE",   "D",  8, 0}, ;
  {"PAID",  "L",  1, 0} }
```

Loading data from a single database

Since an array can have an arbitrary structure it is possible to mimic a database structure in memory with an array. Having the data in an array allows you to access the data at RAM speed rather than disk access speed, and allows you to close the database and release the file handle. If you need numerous small databases for control purposes you can load them once when the application first starts and not worry about the files throughout the rest of the application. For example, you may need lists of valid customer types and stock status codes.

Since we can determine any database structure with the DBSTRUCT() function, it's possible to write a general purpose function to create and fill such an array. The dbf2Array() function expects to be called in such a way that the desired database is available in the current work area.

```
//  Example of call to dbf2Array()
use SOMEDATA new
data_ := dbf2Array()

function dbf2Array
/*
   Return array containing entire contents of database,
       array := { record, record,  ... }
   where each record element is an array of the fields.
       record := { field, field, ... }
*/

local i, r_, a_ := {}

goto top
do while .not. eof()
```

```
      //  Start out with an empty record array
      r_ := {}

      //  Build a list of fields for current record
      for i := 1 to fcount()
       aadd(r_, fieldget(i))
      next i

      //  Add the resulting record to main array
      aadd(a_, r_)

      skip
   enddo

   return a_
```

We may not always want the value of every field to be placed in the array. Let's make the function more versatile by accepting another parameter — an array containing the field names we are interested in. No parameter means all fields.

```
//  Example of call to revised dbf2Array(),
//  load only Name and Phone fields.
data_ := dbf2Array({"Name","Phone"})

function dbf2Array(fields_)
/*
   Return array containing entire contents of database,
       array := { record, record,  ... }
   where each record element is an array of the fields.
       record := { field, field, ... }
   Array of fields may be limited to a list passed as
   a parameter.
*/
local i, r_, f_
local a_ := {}

/*
  If field list not passed,
  assume all fields are desired.
  Create a list of all field retrieval blocks.
*/
if fields_ == nil
  f_ := array(fcount())
  aeval(f_, { |n,i| f_[i] := fieldblock(field(i))})

/*
  Field list was passed,
  create a list of desired field retrieval blocks.
*/
else
  f_ := array(len(fields_))
  aeval(f_, { |n,i| f_[i] := fieldblock(fields_[i])})
endif


// Run through all records in database
goto top
do while .not. eof()
   r_ := {}

   /*
     F_ contains a list of retrieval blocks
     for the desired fields.
     For each field in list,
     add value to record array.
   */
   aeval(f_, { |b| aadd(r_, eval(b)) } )

   // Add record to main array
    aadd(a_, r_)

  skip
 enddo
 return a_
```

This version of dbf2Array() required the use of code blocks in order to best take advantage of Clipper 5's memory management and run-time speed, mainly to avoid the use of macros. See Chapter 8, "Code Blocks," for details.

To make the function even more versatile, let's add a parameter that describes which subset of records we want loaded into the array. No parameter means all records. The best way to handle this is with a code block. The code block should return .t. when we want a record, .f. when we don't.

```
//  Example of call to even fancier dbf2Array(),
//  load Names and Phones, only for the state of Minnesota.
forState := {|| State = "MN" }
data_ := dbf2Array({"Name","Phone"}, forState)

function dbf2Array(fields_, whichRecs)
local i, r_, f_
local a_ := {}

/*
   If field list not passed,
   assume all fields are desired.
  Create a list of all field retrieval blocks.
*/
if fields_ == nil
  f_ := array(fcount())
  aeval(f_, { |n,i| f_[i] := fieldblock(field(i))})

/*
   Field list was passed,
   create a list of desired field retrieval blocks.
*/
else
  f_ := array(len(fields_))
  aeval(f_, { |n,i| f_[i] := fieldblock(fields_[i])})
endif

//  If no code block specified, include all records
if whichRecs == nil
  whichRecs := {|| .t.}
endif
```

```
/*
   For each record in the database,
   check to see if it satisfies the code block.
*/
goto top
do while .not. eof()
  if eval(whichRecs)
  //  Build an array of fields
  //  and add to return array.
    r_ := {}
    aeval(f_, { |b| aadd(r_, eval(b)) } )
    aadd(a_, r_)
  endif
  skip
enddo
return a_
```

Now we have a different sort of problem. Supposedly dbf2Array() fills the **data_** array with field values, but how can we inspect it to be sure? (We're programmers after all, and know these things don't always work the way we intend! The Clipper compiler still uses DoWhatISaid logic, the elusive DoWhatIMeant version is perhaps years away). The following function is used to dump the contents of any arbitrary array to the screen. It uses recursion to doggedly track down each nested subarray. Each "level" is indented a few spaces so you can see the basic structure. DumpArray() uses the level parameter for its own internal use, and you do not need to specify anything. This function is very similar to the ListDir() function presented earlier. DumpArray() adds more documentation as an aid in identifying elements. The function in Listing 9.9 is kept simple to illustrate the basic concept.

Listing 9.9 A function which lists the contents of any array

```
function DumpArray(a_, level)
/*
   List the contents of any array.
   Listing is indented to show nesting of subarrays.
```

```
      This function uses a recursive call to itself.
      Do not specify the level parameter, it is used
     internally during the recursive calls.
  */
  local i
  if level == nil
    level := 0
  endif
  for i := 1 to len(a_)
    ? space(level *4) +str(i,4) +": "
    if valtype(a_[i]) == "A"
      ?? "{..}"
      DumpArray(a_[i], level +1)
    else
      ?? a_[i]
    endif
 next i
 return nil
```

The following is a sample array and what DumpArray() will display. It indicates that the third element is an array, and then goes on to list the contents of that array.

```
test_ := {1,2, {"A","B"}, "Z"}
DumpArray(test_)

//  Resulting display-

1:  1
2:  2
3:  {..}
    1:   A
    2:   B
4:  Z
```

Once loaded into an array you can use ASORT() and ASCAN() to do the same kinds of lookups and loop controlling as you do with .DBF files. Due to the complexity of the array you will need to use the new code block expression features of the ASORT() and ASCAN() functions.

Loading data from multiple databases

While it's certainly useful to load the contents of a single database into an array, we don't have to stop there. By adding more parameters to our dbf2Array() function we can have it track down fields in related databases. We can handle multiple related records by simply adding them as additional elements in an array. The true power of Clipper's arrays becomes very evident indeed.

In order to track down related records and build a single array we need to know the following information.

```
Name of parent database
Fields to include from parent database
Records to include from parent database
Name of related database
Fields to include from related database
Records to include from related database
Key to use to seek into related database
Field(s) in the related database that form the relation
```

An assumption is that the related records all have the same key value, so we can use a simple "while" loop to accumulate the records. This could be replaced by a code block which performs a more complex lookup process. Another simplification is to track down only a single related database. The parameters for the related database could be made into subarrays of a larger array, and the routine could work its way through all the related databases for each parent database record. The less ambitious specifications will allow us to see the fundamental technique without getting lost in features.

Here is an example of a call to this function. The first line indicates we want the ID, Name and City fields from the PARENT database. The double commas (,,) indicate that the "records to include" parameters have been skipped. This is permissible in Clipper 5 and is in fact a very handy feature. The next line indicates we want the Type, Name and Age fields from the CHILD database. The third and fourth lines indicate the two databases are related by the PARENT->ID field into the CHILD->Parent field. We assume both databases are open, and that the CHILD database has an index active with a key expression based on the related PARENT->ID.

```
data_ := dbf2Array("PARENT", {"ID", "Name", "City"},, ;
                   "CHILD",  {"Type", "Name", "Age"},, ;
                   { || PARENT->ID }, ;
                   { || CHILD->Parent == PARENT->ID })
DumpArray(data_)
```

The source code for the new and improved dbf2Array() follows in Listing 9.10. We have simplified the logic by assuming the function will always be used to load a related database. The logic for handling both a single database as well as two related databases obscures the technique, making it needlessly complicated to explain.

Listing 9.10 dbf2Array() revisited: load an array with records from a related database

```
function dbf2Array(parent, pFields_, pRecs, ;
                   child,  cFields_, cRecs, ;
                   key, relation)

local i, r_, rr_, rf_, fP_, fC_
local a_ := {}

/*
   If no list of fields was specified, create an array of
   field value retrieval blocks for all fields
   in database. Otherwise, create array just for
   specified fields.
*/
select (select(parent))
if pFields_ == nil
  fP_ := array(fcount())
  aeval(fP_, { |n,i| fP_[i] := fieldblock(field(i))})
else
  fP_ := array(len(pFields_))
  aeval(fP_, { |n,i| fP_[i] := fieldblock(pFields_[i])})
endif
```

```
select (select(child))
if cFields_ == nil
  fC_ := array(fcount())
  aeval(fC_, { |n,i| fC_[i] := fieldblock(field(i))})
else
  fC_ := array(len(cFields_))
  aeval(fC_, { |n,i| fC_[i] := fieldblock(cFields_[i])})
endif

//  If no code block specified, include all records.
if pRecs == nil
  pRecs := {|| .t.}
endif
if cRecs = nil
  cRecs := {|| .t.}
endif

/*
   For each record in the database, check to see if it
   satisfies the code block.
*/
select (select(parent))
goto top
do while .not. eof()
  if eval(pRecs)
    /*
       Loop through the list of parent fields, building
       an array containing the values of each field for
       the current record.
    */
    r_ := {}
    aeval(fP_, { |b| aadd(r_, eval(b)) } )

    /*
       Seek the key in the child database, build
       an array containing the values for each
       related record.
    */
    rr_ := {}
   select (select(child))
   seek eval(key)
```

```
        do while eval(relation) .and. !eof()

           if eval(cRecs)
              /*
                 Build array of fields in related record.
              */
              rf_ := {}
              aeval(fC_, { |b| aadd(rf_, eval(b)) } )

              /*
                 Add array of fields to related record list.
              */
              aadd(rr_, rf_)
           endif
           select (select(child))
           skip
        enddo

        /*
           Add set of related records to parent record.
        */
        aadd(r_, rr_)

        /*
           Add the parent record value to the return array.
        */
        aadd(a_, r_)

     endif
     select (select(parent))
     skip
  enddo

  return a_
```

Here's an example of a call to dbf2Array() which uses the "which records to include" code blocks. Because we've skipped the "fields to include" array parameters, all fields will be included. The DATA_ array will be filled with records in PARENT where the state is MN, and the CHILD components of the array will be filled only by CHILD records where the value of the Age field is less than 18.

```
data_ := dbf2Array("PARENT", ,, { || State = "MN" }, ;
                   "CHILD",  ,, { || Age < 18 }, ;
                   { || PARENT->ID }, ;
                   { || CHILD->Parent == PARENT->ID })
```

Suppose we use the previous call to dbf2Array() on databases related by ID, where the databases contain family information. Using DumpArray() on the array returned by dbf2Array() displays the results depicted in Figure 9.5. These results are interpreted as "Bob Jones (who's ID is 24334) has a 14 year old son named Tom and a 6 year old daughter named Sue. Mary Smith (ID 52443) of Duluth has no children."

Figure 9.5 Sample output from the DumpArray() function

```
1:  {..}
    1:  24334
    2:  Bob Jones
    3:  Minneapolis
    4:  {..}
        1: {..}
           1:  Son
           2:  Tom
           3:  14
        2: {..}
           1:  Daughter
           2:  Sue
           3:  6
2:  {..}
    1:  52443
    2:  Mary Smith
    3:  Duluth
    4:  {..}
```

A comprehensive example

A final example implements a very complex array structure and demonstrates the use of preprocessor #define directives to make the source code easier to understand.

We've become obsessed with having our applications be completely responsible for creating the data structures they require. We like to be able to hand a client a single EXE file and have it create everything it needs. This takes care of the "oops, we forgot to send them an empty CUSTOMER.DBF file!" And it provides complete documentation for all data structures, right within the application's source code. Our goal is to create and maintain a single array that contains everything the application needs to know about databases, fields, and index keys. We've got the basic design down but are still experimenting with the details. We present here the basic idea which you can modify to suit your own needs.

Let's start from the top and work our way down through the general structure. At the outermost level, the **system_** array contains an element for each database used in the application.

```
system_ := {dbf1_, dbf2_, .. dbfn_}
```

The LEN() of system_ tells you the total number of databases. Each element is itself an array with the following structure.

```
dbf_ := {filename, comment, stru_, index_}
```

Each **dbf_** database array contains a filename, general comment, and two more arrays used to record the field structure and indexes. The **stru_** array contains an array of structure information plus some optional comments about each field. The length of **stru_** tells you the number of fields.

```
stru_ := {fieldname, type, length, decimals, comment}
```

The **index_** element contains an array of index filenames, key expressions and comments about the index.

```
index_ := {filename, key, comment}
```

In practice you establish the **system_** array during the application startup process. The LoadSystem() function is specific to each application, residing in its own source code file.

```
system_ := LoadSystem()
```

We're certain this idea isn't crystal clear when presented at such a level of abstraction, so let's take a look at an example with three databases, in Listing 9.11.

Listing 9.11 Load an aray with database structure and index key information

```
function LoadSystem
/*
   Return array containing complete information
   on all databases, fields and indexes used
   in current application.
*/
local stru_, ntx_
local dbf_ := {}

********************
*** Customer.dbf ***
********************
stru_ := {} ; ntx_ := {}

//  Fields in customer.dbf
aadd(stru_, {"id",      "C",  6, 0, "Customer ID"})
aadd(stru_, {"name",    "C", 30, 0, "Name"})
aadd(stru_, {"address", "C", 30, 0, "Street address"})
aadd(stru_, {"csz",     "C", 30, 0, "City State ZIP"})
```

```
// Indexes for customer.dbf
aadd(ntx_, {"custID",   "id", ;
              "Customers by ID"})
aadd(ntx_, {"custName", "name +ID", ;
              "Customers by Name"})

// Final array of info for customer.dbf
aadd(dbf_, {"customer", "Master customer list", ;
            stru_, ntx_})

*****************
*** Order.dbf ***
*****************
stru_ := {} ; ntx_ := {}

// Fields for order.dbf
aadd(stru_, {"id",       "C", 10, 0, "Order ID"})
aadd(stru_, {"customer", "C",  6, 0, "Customer ID"})
aadd(stru_, {"dated",    "D",  8, 0, "Date of order"})
aadd(stru_, {"sales",    "C",  3, 0, "Salesperson"})

// Indexes for order.dbf
aadd(ntx_, {"ordID",  "ID", ;
             "Orders by ID"})
aadd(ntx_, {"ordCust", "Customer +dtos(Dated)", ;
             "Customer ID by date"})

// Final array of info for order.dbf
aadd(dbf_, {"order", "Customer orders", stru_, ntx_})

****************
*** Item.dbf ***
****************
/*
   Just to be different this DBF_ element is
   constructed directly without using AADD() for
   individual array components. We feel this
   method is more difficult to get right the
   first time and harder to maintain.
*/
```

```
aadd(dbf_, {"item", "Customer order line items", ;
  { ;  //  Structure
    {"order",  "C", 10, 0, "Order ID"}, ;
    {"invent", "C",  8, 0, "Inventory item"}, ;
    {"qty",    "N",  4, 0, "Quantity"} ;
  },;
  { ;  //  Index
    {"itemOrd", "Order +Invent", ;
     "Order items by inventory code"} ;
  } ;
})

*** The DBF_ array now contains complete system info.

return dbf_
```

Given such an array we can create functions to perform a variety of useful services.

```
//  Create DBF structures for entire application
SysCreate(system_)

//  Create indexes just for CUSTOMER.DBF
SysIndex(system_, "Customer")

//  List everything we know about the application,
//  then just for ORDER.DBF.
ListStru(system_)
ListStru(system_, "Order")
```

Before writing these functions, let's create a preprocessor #include file that will help make the source code easier to read and understand.

```
/*
   DBFSYSTEM.CH
   The main system array has four elements
*/
#define SYS_DBFNAME    1
#define SYS_COMMENT    2
#define SYS_STRU       3
#define SYS_NTX        4

//  Each SYS_STRU element is an array of five elements
#define STRU_NAME      1
#define STRU_TYPE      2
#define STRU_LEN       3
#define STRU_DEC       4
#define STRU_COMMENT   5

//  Each SYS_NTX element is an array of three elements
#define NTX_NAME       1
#define NTX_KEY        2
#define NTX_COMMENT    3
```

These #define directives are made available to all functions in the source code file through the use of an #include directive.

```
#include "dbfsystem.ch"
```

On with the source code. We begin by writing the function that creates database structures based on the contents of the **system_** array passed to it. See Listing 9.12.

Listing 9.12 Create databases based on contents of an array.

```
function SysCreate(sys_, which)
/*
   Given a system structure array and an optional
   filename, create database(s).
*/
```

```
local n
if which <> nil
  n := ascan(sys_, ;
       { |d_| upper(d_[SYS_DBFNAME]) == upper(which)})
  if n > 0
    SysCreate1(sys_, n)
  else
    //  Error handler?
  endif

else
  for n := 1 to len(sys_)
    SysCreate1(sys_, n)
  next n
endif
return nil

static function SysCreate1(sys_, n)
local dbstru_ := {}, f
for f := 1 to len(sys_[n, SYS_STRU])
   aadd(dbstru_, {sys_[n, SYS_STRU, f, STRU_NAME],;
                  sys_[n, SYS_STRU, f, STRU_TYPE],;
                  sys_[n, SYS_STRU, f, STRU_LEN ],;
                  sys_[n, SYS_STRU, f, STRU_DEC ]})
next f
dbcreate(sys_[n, SYS_DBFNAME], dbstru_)
return nil
```

In SysCreate(), the call to ASCAN() searches the filename portion of the system array looking for the specified name. Since the system array contains only arrays we need a code block to tell ASCAN() exactly how to perform the search within the array structure. At the same time we have ASCAN() make the search insensitive to character case.

The function SysCreate1() is declared static so we can make the main SysCreate() function easier to understand. A regular function is visible to the rest of the application, so we "protect" SysCreate()'s sub-function by making it static and consequently visible only to the other functions in the source code file. This is a way to tell anyone looking at the code that they are only supposed to make calls to SysCreate() and not SysCreate1(). Listing 9.13 continues with the SysIndex() function.

Listing 9.13 Create index files based on contents of array

```
function SysIndex(sys_, which)
/*
   Given a system structure array and an optional
   database filename, create associated index(es).
*/

local n
if which <> nil
  n := ascan(sys_, ;
       { |d_| upper(d_[SYS_DBFNAME]) == upper(which)})
  if n > 0
    SysIndex1(sys_[n, SYS_DBFNAME], sys_[n, SYS_NTX])
  else
    //  Error handler?
  endif

else
  for n := 1 to len(sys_)
    SysIndex1(sys_[n, SYS_DBFNAME], sys_[n, SYS_NTX])
  next n
endif
return nil
```

```
static function SysIndex1(dbfname, ntx_)
local i, key
for i := 1 to len(ntx_)
  use (dbfname) new
  key := ntx_[i, NTX_KEY]
  index on &key. to (ntx_[i, NTX_NAME])
  use
next i
return nil
```

The code for calling the indexing routine is very similar to SysCreate(). The main difference is that in SysCreate() we passed the entire system array to SysCreate1(), and broke the array into component pieces down there. The main indexing routine took a different approach and determined the filename and isolated the array of indexes before calling Index1(). The two methods are equivalent. We used different methods for illustration only. In practice you should use a consistent method to help make maintenance easier.

The final function to write is ListStru(), which documents each database in detail. Listing 9.14 contains the source code. If you can read and understand this function you can be confident you have a very good grasp of multiple dimension arrays. If not, don't be depressed. We use such arrays all the time and we still get confused sometimes.

Listing 9.14 Print a listing of database structure and index key information based on contents of an array

```
function ListStru(sys_, which)

local n
if which <> nil
  n := ascan(sys_, ;
       { |d_| upper(d_[SYS_DBFNAME]) == upper(which)})
  if n > 0
    ListStru1(sys_, n)
  else
```

```
      //  Error handler?
    endif

  else
    for n := 1 to len(sys_)
      ListStru1(sys_, n)
    next n
  endif

  return nil

  static function ListStru1(sys_, n)
  local i
  ?
  ? "Database: " +upper(rtrim(sys_[n, SYS_DBFNAME]))
  ?? ", " +sys_[n, SYS_COMMENT]
  ?

  for i := 1 to len(sys_[n, SYS_STRU])
    ? str(i, 4) +space(2)
    ?? padr(sys_[n, SYS_STRU, i, STRU_NAME], 10)
    ?? space(2) +sys_[n, SYS_STRU, i, STRU_TYPE]
    ?? space(2) +str(sys_[n, SYS_STRU, i, STRU_LEN], 3)
    //  Only numeric fields need the decimals listed
    if sys_[n, SYS_STRU, i, STRU_TYPE] = "N"
      ?? "." +str(sys_[n, SYS_STRU, i, STRU_DEC], 1)
    else
      ?? space(2)
    endif
    //  Field comments are optional
    if len(sys_[n, SYS_STRU, i]) = STRU_COMMENT
      ?? space(2) +sys_[n, SYS_STRU, i, STRU_COMMENT]
    endif
  next i

  ?
  for i := 1 to len(sys_[n, SYS_NTX])
    ? space(3)
    ?? upper(rtrim(sys_[n, SYS_NTX, i, NTX_NAME]))
  ?? ", " +sys_[n, SYS_NTX, i, NTX_COMMENT]
  ? space(5) +"key: "
  ?? sys_[n, SYS_NTX, i, NTX_KEY]
next i
?
return nil
```

If the **system_** array were still around from the previous example, the result of a call to **ListStru(system_, "customer")** will look like the following.

```
Database: CUSTOMER, Master Customer List

   1  ID          C   6    Customer ID
   2  Name        C  30    Company Name
   3  Address     C  30    Street Address
   4  CSZ         C  30    City, State ZIP

   CUSTID, Customers by ID
     key: ID
   CUSTNAME, Customers by Name
     key: Name +ID
```

This concept lends itself to a variety of applications. You can pass around everything known about your application's databases and indexes using a single array. Only a single version of your general purpose maintenance routines need to be maintained if they always operate based on such a **system_** array.

Loading unknown database structures

Listing 9.15 contains a routine that creates everything (except the comments and indexes) for a **system_** array from a collection of database files, rather than requiring you to define the structure via source code. We prefer the "fresh from the source code" approach, but can see circumstances where you can't define the structure in advance.

Listing 9.15 Load an array with database structure information based on the DOS directory

```
#include "DIRECTRY.CH"
function DOS_Load
/*
   Load array containing database and field information
   directly from the contents of the DOS files, rather
   than using a source code routine.
*/
local i, name_, dbf_ := {}
name_ := directory("*.DBF")
for i := 1 to len(name_)
  use (name_[i,F_NAME]) new
  aadd(dbf_, {name_[i,F_NAME], "", dbstruct(), {}})
  use
next i
return dbf_
```

The following is an example of a call to DOS_Load().

```
function Main
system_ := DOS_Load()
//
//  Remainder of application
//
return nil
```

The DOS_Load() function takes the array of .DBF file information returned by DIRECTORY() and uses it to load an array of database filename and structure information. The description of each database (the second element of the main array), the comments about individual fields (the fifth element of the structure array), and index information (fourth element of the main array) cannot be determined from DOS directory information alone, so the ListStru() function is considerably less useful.

If you had a file-naming convention that associated indexes with databases you could also automatically load the index names and key expressions. Depending on the index key expressions and the way you name fields it may even be possible to associate an index with a database via the index key alone. This takes advance planning and rigorous attention to a naming convention that always results in a unique association of databases and index keys.

Saving and restoring arrays

Despite Clipper's almost overwhelming implementation of arrays it manages to omit one critical feature, namely, the ability to save and restore the contents of an array directly to and from disk storage. The regular Clipper storage mechanisms, database (.DBF) and memory (.MEM) files, are very inefficient and limited in the kinds of arrays they can accommodate, and even then, only after some tortuous programming.

In the final section of this ambitious chapter we provide solutions for direct array storage and retrieval: SaveArray() and RestArray(). These functions have been designed to handle any Clipper array structure and contents. A powerful programming technique, called recursion, allows us to process deeply nested arrays the same way we process simple arrays with single dimensions.

Since Clipper has a complete set of data type functions we can store any data type to a file and read it back in again. The only exception is the code block data type. A code block is really more of a reference to program code sitting in memory than an actual sequence of digits or characters or logical values, so it is not possible to transfer a code block outside of the application. You can, however, store the source code for a code block as a character string and use the macro compiler operator, &(), to produce a code block.

```
//  An array with two code blocks as elements.
//  Can't save this array outside of the application.
a_ := { { || Test1() }, { || Test2() } }
```

```
// An array of two character strings.
// The strings form code blocks.
// This array can be saved outside of the application.
b_ := { "{ || Test1() }", "{ || Test2() }" }

// Compile and run the code block defined by
// the source code in array element number one.
c :=&(b_[1])
eval(c)
```

The SaveArray() function will write a NIL value should it encounter a code block. This is done to preserve the original structure of the array and not because NIL is a reasonable approximation of the code block.

Saving an array to a file

Saving an array to a file is accomplished by a call to the SaveArray() function. The syntax is illustrated below.

```
n_ := {1,2,3, {4,5,6, {7,8,9}}}
if SaveArray(n_, "NUMS.ARY")
  ? "Array-save process was successful."
else
  ? "An error occurred during the array-save process."
endif
```

You pass the array to save and the name of the target file. The target file will be created if it does not exist and overwritten, without warning, if it does exist. As the above example indicates, SaveArray() returns a logical value signaling the success or failure of the operation.

Restoring an array from a file

The file created by SaveArray() is of limited value unless there's a way to read it back in again, and that's done through a call to the RestArray() function. Once again an example will clearly illustrate the syntax.

```
x_ := RestArray("NUMS.ARY")
if len(x_) > 0
  ? "Array restored successfully."
else
  ? "An error occurred during the array-restore process."
endif
```

You pass the name of the file containing the array. The function returns a reference to the array it creates.

Source code

The SaveArray() and RestArray() functions are actually just the publicly visible interfaces to other functions that do all the work. (This technique is discussed in more detail in Chapter 12, "Program Design.) The source code for both sets of functions is found in Listings 9.16 through 9.19. The comments within the source code listings completely document the design and programming techniques.

Listing 9.16. The SaveArray() function. Writes an array to a disk file.

```
function SaveArray(a_, fileName)
/*
   General-purpose function for saving an array to a disk
   file. This is only the public interface. The real work is done
   by recursive calls to the ElementOut() function.

   Pass an array (or a reference to an array) and the name of the
   file to store the array.

       SaveArray({1,2,3,4}, "NUMS.ARY")

   This function returns .t. if successful, .f. if not.
*/
```

```
local cnt := len(a_)
local success := .f.
local handle := fcreate(fileName)

if handle != -1
   success := ElementOut(handle, a_)
   fclose(handle)
endif

return success
```

Listing 9.17 The ElementOut() function. Handles output for a single array dimension

```
static function ElementOut(handle, a_)
/*
   Given a file handle and an array, write the contents of the
   array to the file in the following form.

      LL T WW E... T WW E...

   Where LL is a two byte integer representing the number of
   elements in the array, T is a character representing the data
   type of the first element, WW is a two byte integer
   representing the width of the element; followed by repetitions
   of that basic pattern.

   Nested arrays are handled by calling ElementOut() whenever an
   array is encountered within the elements in the array
   currently being processed. (Known as recursion.)  The LL length
   bytes are written following an "A" data type and the process
   gets repeated. Isn't recursion wonderful?

   This is a static function and therefore not visible to any
   functions outside of the source code file containing it.
   All calls from the outside must be made to SaveArray().
*/

local success := .t.
local i, buffer
local cnt := len(a_)
```

```
// Write the overall array size.
fwrite(handle, i2bin(cnt))

// Process each element in the array.
for i := 1 to cnt

     /*
      Special handling for the nil and code block data types.
      Both will be labeled type "Z" and for consistency with
      the other data types, will have a width of one and an
      element value of "Z". However, a NIL will be placed in
      the array when it comes time to load it from the file.
     */
  if (a_[i] == nil) .or. (valtype(a_[i]) == "B")
       buffer := "Z" +i2bin(1) +"Z"

  else
     /*
       Each element is encoded as follows.
           Data type:  C,D,L,N.
               Width:  Number of characters needed.
       Element Value:  String version of the value.
     */

    buffer := valtype(a_[i])
    do case
    case buffer == "C"
      buffer += i2bin(len(a_[i]))    //  Width of the string
      buffer += a_[i]

    case buffer == "D"
      buffer += i2bin(8)             //  Dates are 8 wide
      buffer += dtoc(a_[i])

    case buffer == "L"
      buffer += i2bin(1)             //  Logicals are 1 wide
      buffer += if(a_[i], "T", "F")
```

```
         case buffer == "N"
            //  Convert number to string, trim spaces, and
            //  calculate width of number based on the string.
            buffer += i2bin(len(ltrim(str(a_[i]))))
            buffer += ltrim(str(a_[i]))

         otherwise
            //  Type "A" for arrays will be handled
            //  after we write the type.
         endcase
      endif

      //  Write the buffer, constructed above, to the file.
      if fwrite(handle, buffer, len(buffer)) != len(buffer)
         success := .f.
         exit
      endif

      /*
         If this is a nested array, it's recursion time!

         Call ElementOut() again, it will append a series
         of types/widths/values to the current file.
      */
      if left(buffer, 1) == "A"
         ElementOut(handle, a_[i])
      endif
   next i

   return success
```

Listing 9.18 The RestArray() function. Restores an array from a disk file

```
function RestArray(fileName)
/*
   General-purpose function for restoring an array from a disk
   file. This is only the public interface. The real work is done
   by recursive calls to the ElementIn() function.
```

```
    This function expects the name of a file created previously by
    the SaveArray() function. It reads the file to construct the
    array that was saved.

      n_ := RestArray("NUMS.ARY")

    This function returns a reference to the array it creates.
*/

local handle, a_ := {}

if (handle := fopen(fileName)) != -1
   ElementIn(handle, a_)
endif
fclose(handle)

return a_
```

Listing 9.19 The ElementIn() function. Handles restoring of a single array dimension

```
static function ElementIn(handle, a_)
/*
  Given a file handle and a reference to an array, read elements
  from the file and add them to the array.

  The file must be created by the SaveArray() function and be
  structured as described in the SaveArray() comments.

  Nested arrays are detected and sub-arrays stored in the main
  array as needed. The file reading for a sub-array is
  accomplished through another call to ElementIn(). This is
  known as recursion.

  This is a static function and therefore not visible to any
  functions outside of the source code file containing it.
  All calls from the outside must be made to RestArray().
*/
```

```
local buffer, i, cnt, iLen, iType

//  Read the overall array size
buffer := space(2)
if fread(handle, @buffer, 2) = 2

  //  Process each array element stored in the file.
  cnt := bin2w(buffer)
  for i := 1 to cnt

    //  Read the element's data type.
    //  If element is a nested array - recursion time!
    //
    if (iType := freadstr(handle,1)) = "A"
      aadd(a_, {})
      ElementIn( handle, a_[ len(a_) ] )

    else
      //  Read the length of the element.
      buffer := space(2)
      if fread(handle, @buffer, 2) = 2
       iLen := bin2w(buffer)

        //  Read the actual element.
        buffer := space(iLen)
        if fread(handle, @buffer, iLen) = iLen

          //  Convert from string to specified data type.
          do case
           /*
             Note that we can't save code blocks. If you
              attempted to save one from an array, we will have
              empty space and thus must add a NIL to serve
              as a placeholder.
```

```
            */
            case (iType == "B") .or. (iType == "Z")
               aadd(a_, nil)
            case iType == "C"
               aadd(a_, buffer)
            case iType == "D"
               aadd(a_, ctod(buffer))
            case iType == "L"
               aadd(a_, (buffer == "T"))
            case iType == "N"
               aadd(a_, val(buffer))
            endcase
         endif
      endif
   endif
next i
endif
return nil
```

Summary

You have been introduced to arrays and shown the power and flexibility they offer. We've only scratched the surface of their possible uses. Based on your understanding of how arrays are constructed and manipulated you will be able to apply arrays to almost every other facet of Clipper programming.

We've seen how Clipper 5's ability to create and manipulate arrays with multiple dimensions and a mixture of data types can be put to use to solve real-world problems. The Clipper language has the features needed to perform extremely

Reexamination 90/005,727

The Original

Page 372 of Part II

Is Missing

Initial : AR
Date : 6-8-2000

CHAPTER 10

Debugging

The debugger bundled with Clipper 5 is completely new and takes a different approach from the Summer '87 version. It is a major improvement in every respect. On the whole, it makes the task of debugging less burdensome and at times — dare we say it? — even enjoyable.

This chapter is devoted to providing an overview of the new debugger's features so you know what tools you have at your command. It closes with some "classic combinations" of features that may not be immediately obvious to the first-time user. Along the way you should pick up some useful tips and techniques, including how to avoid some initial frustration.

The debugger has a number of what we feel are either obvious or at least easily learned features that are not in need of elaboration. Examples of these are loading and scrolling through files, searching for text with find-next and find-previous, and other features that any programmer who ever used a text editor will easily figure out. Also not covered are the important but conceptually simple procedures for moving and resizing windows and changing the debugger's color scheme. All these features can be learned more quickly by just plain doing rather than by reading.

Note: In the interest of brevity we use the term **user-defined function**, and more often just **function**, rather than the more accurate (but verbose) "user-defined function or procedure". The debugger works equally well with functions or procedures. It does not operate on the internal workings of Clipper itself, so in this context the word function applies only to those that you write yourself and not to Clipper's built-in functions.

Summer '87 comparison

The Summer '87 debugger is an object file, DEBUG.OBJ, that has to be linked into your application's EXE along with the rest of your code. Consequently it occupies about 40KB of EXE file space and 35KB RAM. To invoke the debugger you must load and execute your application, and then press ALT-D or make a direct call to the ALTD() function within your source code.

In Summer '87, having to link DEBUG.OBJ into your application has a number of drawbacks. In addition to the memory and file size issue, you have to make a conscious effort either to link the debugger when you need it or to always have it linked while your application is under development and suffer a constant drain on link time and execution speed. Execution speed will be degraded even if you don't use the debugger; as long as it's linked it will be demanding attention from the application.

In Clipper 5, the debugger is a stand-alone utility, CLD.EXE, that contains all the debugging tools. Every application you link, regardless whether you thought you might need the debugger or not, can be used with the CLD utility. To use the debugger you simply specify the name of your application's .EXE file as a DOS command line parameter. For example, the following DOS command runs TEST.EXE within the debugger.

```
cld test
```

The CLD.EXE utility loads your application and executes it. Since CLD was loaded into DOS first it can perform all sorts of almost magical services for you. Well, they will seem magical to Summer '87 programmers, but not to other programmers who have seen Microsoft's CodeView or Borland's Turbo Debugger. The Clipper 5 debugger is influenced by these two extremely powerful debuggers.

Note: The Clipper 5 debugger can also be linked directly into an application for situations that demand such a configuration. The nice thing is, linking is optional. For most debugging sessions the CLD utility is the most efficient method.

What the debugger needs

Before getting started with operational details let's take a brief look at what the debugger expects from you. To get the most out of the debugger you need to supply it with several important kinds of information. It can work with less than all of the information, but with varying degrees of success.

Symbols

Many of Clipper 5's internal structural changes result in your code being optimized, particularly with respect to the "symbols" you create. Symbols are anything for which you make up a name: memory variables, function names and so on. Many of these symbols are converted to direct references to actual memory locations and consequently the name you assigned disappears. The .EXE file does not always contain the symbol name. Examples of this are **local** and **static** memory variables. The CLD utility will not be of much use to you if you can't see your original symbol names. The Clipper 5 compiler can, however, be instructed to keep the original information intact so the debugger can make use of it. This is done via the /B compiler switch.

```
clipper test /b
```

Using the /B switch will slightly increase the size of the object file (since it contains additional debugging information) and consequently the size of the .EXE. Using a dynamic overlay linker like RTLink makes this a non-issue, so we have included the /B switch in our standard Clipper compiler environment command (as mentioned in Chapter 1).

```
set clippercmd=/a /b /m /n /p /v /w
```

This ensures that debugging information is always available. If space is critical you can always recompile all objects for the application and link one final time before distributing the application EXE file.

Object code

Since special debugging information is stored in the object file (when compiled with the /B switch), the debugger needs access to the object files (.OBJ) while executing your application. The debugger can get very confused if you have recompiled any object files since the last time the .EXE was linked.

Source code

The debugger is termed a "source level debugger," meaning it can follow your source code while your application is executing. Not surprisingly the debugger needs to be able to read the source code files (.PRG) while running your application. Things will be totally out of whack if you have edited the source code since the associated object file (.OBJ) was last compiled. The debugger may appear to work but will eventually lose track of the correct source code line.

Line numbers

In order to track your application's progress through the original source code the debugger needs to refer to line numbers. Clipper stores such line numbers in the object code by default. If you suppress line numbers with the /L compiler switch you will prevent the debugger from tracking the source code. Since you have a dynamic overlay linker (.RTLink) there is very little gained by compiling without line numbers. As with the /B switch you can always recompile and link after you no longer need the line numbers for the debugger.

Removing line numbers has an additional negative side effect of making run-time errors less informative, because you'll be told only the name of the function where the error occurred, and no line number.

Preprocessor output

The debugger is capable of tracing the output from the preprocessor in addition to your original source code. Since the compiler never actually sees your source code (only the preprocessor deals with source code) there are certain classes of errors that cannot be understood by viewing the original source code. The preprocessor may have introduced the error, perhaps through a bad #define, #command, or #translate directive. If you ever need to trace the preprocessed version of your source code you will need to compile with the /P switch. Creating preprocessed output files, which end with a .PPO extension, is a time- and disk space-consuming task. Depending on the speed of your computer and the amount of disk clutter you will put up with, you may not want to have this done automatically via SET CLIPPERCMD.

Starting from DOS

To use the debugger, simply run the CLD.EXE program and supply it the name of a Clipper executable file. The following line starts the debugger with TEST.EXE as the target Clipper application to work with.

```
cld test
```

The debugger has a number of DOS command line switches that control its operation. These options are presented in the order they must occur; CLD is one of the few programs where the sequence is significant.

/43	43-line mode, or
/50	50-line mode, or
/S	Split screen into two halves
@Script	Run debugger commands found in file
AppName	Name of Clipper application to debug
AppParams	Parameters to pass to Clipper application

We'll discuss the use of the script option later in this chapter. The debugger options must be specified ahead of the application name. Any parameters following the application name will be passed along to the application.

Try the split screen feature if you have a monitor capable of displaying at least 43 lines. You'll never want to go back to debugging without a split screen again. Here's an example.

```
cld /s maxibrow customer
```

Translation: Debug MAXIBROW.EXE in split screen mode. Pass "customer" as a parameter to the MAXIBROW program.

The screen mode parameters are mutually exclusive, and if you specify more than one only the last will be used. The split screen mode will use either 43 or 50 line mode, depending on what your video adapter card is capable of doing.

Linking the debugger directly

The Clipper 5 debugger can also be linked directly into the application .EXE file via the CLD.LIB library file. Even if linked this way, the debugger still requires the /B switch when compiling. All other compiling issues discussed previously apply as well.

If linked, the debugger must be invoked manually by pressing Alt-D or by calling the ALTD() function from within the application code itself. See the discussion on the ALTD() function for details.

Overview of debugger features

A complete discussion of every debugger feature could fill several chapters and still not cover everything in detail. We will highlight some of the more important features and leave the rest to your experimentation. We strongly urge you to spend as much time as you can playing with the debugger before you actually need it in a panic situation. This utility can be very frustrating until you have mastered the basic concepts. You don't want to be learning the fundamentals with a nasty UNDEFINED IDENTIFIER problem at 5:00 p.m. on the Friday your application is due!

The debugger is also an excellent environment for learning how Clipper works. Rather than always waiting for a confusing problem or run-time error, fire up the debugger and watch what happens with small programs. You'll learn more watching your code execute in the debugger than if you relied only on what the application displays on screen.

Once the debugger is finished loading it will halt at the first executable statement in your application. What happens from there is the subject of the rest of this chapter.

General navigation

You can access most of the features of the debugger three different ways. The debugger command line is universally available for almost all features and is discussed in more detail later in this chapter. The debugger main menu can be accessed via the ALT key and the letter corresponding to a menu choice, like ALT-F for File. Frequently used features have function key short-cuts, like F9 for setting or clearing a breakpoint. Some features pop up a dialog box when they need additional information.

Table 10.1 lists the function keys as they are used in the debugger.

Table 10.1 Function keys

F1	Display help screens
F2	Zoom current window (toggle)
F3	Repeat last command on command-line
F4	Application screen (toggle)
F5	Resume execution
F6	Work area information
F7	Run to cursor
F8	Step one line
F9	Set breakpoint (toggle)
F10	Step over function

When multiple windows are displayed, the Tab and Shift-Tab keys are used to select different windows. Windows can be moved, sized, and even minimized to give you a better view of the underlying application.

Unless you are in a dialog box, any typing you do will be placed in the command line window independent of which window is actually selected. This may take some getting used to. Keep in mind that the debugger displays things in the windows based on your menu choices or the commands you type. You never type directly into any window, unless it pops up a dialog box. Once you're familiar with the nature of the command window you'll appreciate how efficient it is. You do not have to tab over to the command window — just type from wherever you happen to be at the time.

Switching screens

If you are running the debugger in split screen mode (the recommended way if your video adapter card can handle it), you don't need to concern yourself with switching between the debugger screen and the application screen — both will be completely visible at all times. If you can't run the debugger in split screen mode there are a number of other features designed to make your debugging more enjoyable.

First, the F4 key is always available to switch between the debugger screen and the application screen. As you work your way through the source code you can tap F4 to see what's going on "outside." Another press of the F4 key will send you back to the debugger screen. You can bounce back and forth as often as needed, without disturbing any of the debugger settings.

A better alternative is to move and resize the various windows to give a decent view of both the debugger's and your application's screens. Being able to see both screens makes it much easier to keep track of what's going on. All windows have built-in features for making this a simple process. Once you've configured the display to your liking you can save it to a script file. Use of script files is also covered in more detail later in this chapter.

Working with source code

When you're first getting started with the debugger, its main purpose will seem to be to step through your source code a line at a time. This is indeed an important feature and a great place to get started. However, don't stop there! Once you get the hang of the other debugger features you'll find you seldom need to "walk" through your code in this way.

When the debugger first starts up, the source code window will display the first executable line of the first function in your application. Note that some statements are not considered to be executable lines. For example, **local** and **static** declarations will be skipped, along with any preprocessor directives. Comments are skipped as well. The debugger may appear to have leapt well into your application, when actually it's just stopping at the first executable line. An exception to this rule is when you combine a declaration with an assignment operation. The debugger will stop at such lines.

```
#include "inkey.ch"  //  Debugger does not stop on this line.
#define DEBUG        //  Not here, either.
static a, b, c       //  Nope.
local x := 0         //  But, it does stop here.
```

Moving through the source code

There are two kinds of source code movement.. The debugger keeps track of the line that's about to be executed and highlights it in inverse video. When the source code window is selected you can scroll around through the code, but the debugger always stays parked on the line about to be executed. You move the debugger's highlight any of the following ways:

- **Stepping line by line.** Pressing the F8 key causes the debugger to execute the highlighted line and proceed to the next. If the line was a call to a user-defined function the debugger branches to that routine and highlights the first executable line there. If the function is located in a different source code file, that file is opened automatically. (A message is displayed if no source code can be located). If the line requires a code block to be evaluated, the debugger jumps to the place where the code block was either defined (if it's a literal code block) or where it was compiled via the &() macro compiler. You can turn off this code block tracing feature under the Options menu.

- **Stepping over functions.** Sometimes you are only interested in tracking the source code in a single function and don't want to trace the execution of every single line in the application. The Clipper debugger calls this "tracing", and conveniently ties it to the F10 key. Pressing F10 is similar to F8 in that the debugger will advance to the next executable line; however, it will not track function calls outside of the current function.

- **Run to cursor.** Rather than repeatedly whacking the F8 or F10 keys to advance the source code highlight to a line in which you are interested, the F7 key can be used to allow the application to run at full speed until it hits the line at which you placed the cursor. (You must move the source code cursor to an executable line before the program pointer will advance.) The following example clearly illustrates the usefulness of this feature.

```
n := 0                  //  Current debugger highlight is here...
for i := 1 to 1000
  n += Foo(i)
next i
? n                     //  ...run to cursor, here!
```

- **Resume execution.** Frequently you'll find it much easier to just let your application run as it normally does and then stop it when it does (or is about to do) something interesting. There's a wide array of debugger features you can use to help stop the application at opportune times. Just hit the F5 key to give control back to your application. It will execute without pausing. The debugger can be invoked once again by pressing ALT-D. We'll cover other techniques, like breakpoints and tracepoints, later in this chapter.

- **Restarting.** The debugger has a Restart option (found under the Run menu) for starting your application over from scratch. This is handy if you had to run the application to the point it hit a fatal error. Simply note where the error occurred and issue a restart. This time you can use the debugger to help determine what went wrong.

Sharing screens and animation

The debugger has a mode of operation called **Animate**, where it automatically executes source code lines one at a time with an adjustable pause between each line. **Animate** is found under the Run menu, as is the animation speed control. Animation speed is measured in tenths of a second to pause between lines, with zero being no pause at all.

This screen can be very confusing. By default, the debugger will switch to the application automatically after each line is executed. You can control this behavior with the **Exchange Screens** option under the Options menu. When OFF, the debugger will only switch to the application screen when input is required. You can eliminate even this screen switch via the **Swap on Input** option, also found under the Options menu. With both these options set off the application screen will not be displayed. Of course, you can interrupt the animation with ALT-D and press the F4 key to see the application screen at any time.

Animation is useful when you want to slow down the execution of your application and monitor it as it trundles along. If you can't run in split screen, however, the constant screen switching will likely drive you crazy.

The best way to run Animation is with the source code window made small enough and moved so that the application screen is visible. The other windows can be sized and moved as well, resulting in a reasonable balance between information on the application and the debugging information covering some of it.

Likely problem areas

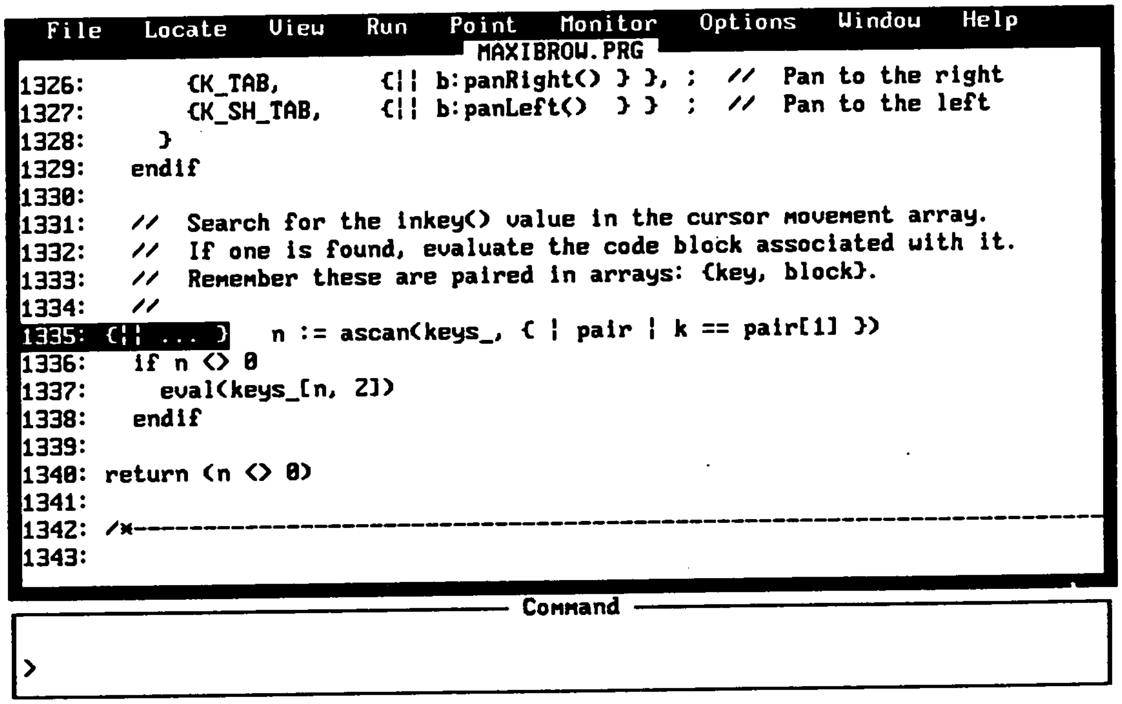
There are several kinds of source code that are likely to cause confusion in step-through operations.

- **Preprocessor side effects.** Any lines directed to the preprocessor are, by their very nature, not going to be dealt with very well at run-time. The compiler, and consequently the run-time application, never see these lines. Manifest constants will not be visible anywhere but in the original source code. You can't test or alter the values of manifest constants because the **names** don't really exist in the .EXE; the preprocessor swapped the name with the value and the compiler never even saw it. Directives for including header files are also potentially troublesome. You can always use the File-Open feature to view the actual header file. You can have the debugger display the preprocessed code (selected under the Options menu), in which case you'll see the source code both before and after the preprocessing.

- **Multiple statements in single line.** The preprocessor routinely sticks multiple statements on the same source code line (using a semicolon to separate them) and consequently so can you. In limited circumstances this may be desirable, but be aware that the debugger treats the entire line as a single entity and can't individually process each statement. You are much better off keeping each statement on its own line. You gain nothing as far as run-time speed is concerned but you lose debugging capabilities.

- **Code blocks.** Code blocks allow you to write code in one place and ship it off for execution somewhere else. However, code blocks are evaluated in the context of where they were *defined*, not where they are executed. To the debugger this makes code blocks look like a function call. The debugger will trace execution back to the source code line where the code block was defined. You can disable this behavior via the *Code Block Trace* choice on the Options menu. The preprocessor will convert some otherwise innocent-looking commands to include code blocks, so don't be surprised to see the code block marker, {||...}, displayed next to source code lines that don't appear to have anything to do with code blocks. Turning on the preprocessed code display option will reveal the hidden code blocks.

Figure 10.1 Code block trace



```
      {K_TAB,      {|| b:panRight() } }, ;  //  Pan to the right
      {K_SH_TAB,   {|| b:panLeft()  } }  ;  //  Pan to the left
    }
  endif

  //  Search for the inkey() value in the cursor movement array.
  //  If one is found, evaluate the code block associated with it.
  //  Remember these are paired in arrays: {key, block}.
  //
{|| ... }  n := ascan(keys_, { | pair | k == pair[1] })
  if n <> 0
    eval(keys_[n, 2])
  endif

return (n <> 0)

/*------------------------------------------------------------------
```

```
>
```

Inspecting things

Once you get over the thrill of stepping your way through your source code (which may take quite a while, it's great fun!) you should turn your attention to what is arguably the most useful feature of the debugger — its ability to inspect and alter the value of almost everything in your application.

The ? command

To see the value of any variable you can use the question mark command in the command window. The following displays the current value of x.

```
? x
```

This is best used in situations where you know the name of the variable and only want a quick peek at the value. If you don't know the name, see the later discussion on the Monitor feature. If you want to keep checking on the value, you're better off setting a watchpoint, which we will discuss in a moment.

Results of expressions

The ? command is actually displaying the results of Clipper expressions, not just simple variable names. The result of any valid Clipper expression can be displayed, providing all functions referenced have been linked into the application's .EXE. If you want to see the last 20 characters of a 1,000 character string, you can call RIGHT().

```
? right(longstring, 20)
```

You can perform mathematical calculations, data type conversions, anything at all so long as it is an acceptable Clipper expression.

```
? val(substr(data, 2, 7)) * (factor /10)
```

Since variable assignment is considered an expression in Clipper 5, you can also use the question mark command to assign values to memory variables. You can even create new variables this way. The following example either assigns **d** a new value (if it currently exists and is in scope) or creates a new variable called **d** and gives it a **private** scope (if **d** does not exist).

```
? d := date()
```

Note that you must use the inline assignment operator (:=) to make assignments. Otherwise the debugger thinks you are asking it if the variable is currently equal to the value. The following will display .T. or .F. depending on the value of **d**.

```
? d = date()
```

The ? command is capable of evaluating any legal expression, so you can do some amazing things with it. See the section "Adding Your Own Features," later in this chapter.

Watchpoints

Watchpoints open a window at the top of the screen that displays the current value of the expression you specify. Watchpoints can be set by selecting **Watchpoint** from the Point menu, or more efficiently by entering them at the command line. The following watchpoint will continuously display the current value of the **counter** variable:

```
wp counter
```

For an example that illustrates a constantly changing value, you can set a watchpoint on the PROCLINE() function. Enter the following at the command line.

```
wp procline()
```

While the program executes, the current line number will be displayed in the watch window. Any expression can be used as a watchpoint. Typically you set a watchpoint on a variable name or expression that you want to keep track of, especially when the value is independent of the function that's currently executing. A watchpoint is constantly evaluated, even when the expression is irrelevant to the function being executed.

See the "Classic Combinations" section of this chapter for some additional tips and techniques.

More efficient variable inspection

The fastest way to see the values of many variables is to use the Monitor menu and select the combination of storage classes in which you are interested. The currently active variables of each class will be displayed in the watch window. Remember that if more variables are displayed than can fit in the window you can tab over to the window and scroll around. You can also change the size of the window if you want to see large numbers of variables. A handy alternative is the F2-Zoom key, which you can use to toggle a full screen view of the Watch or Monitor windows.

See the discussion on the Callstack, and also the "classic combination" of Callstack and Monitor, later in this chapter, for some very powerful debugging tips and techniques.

Altering values

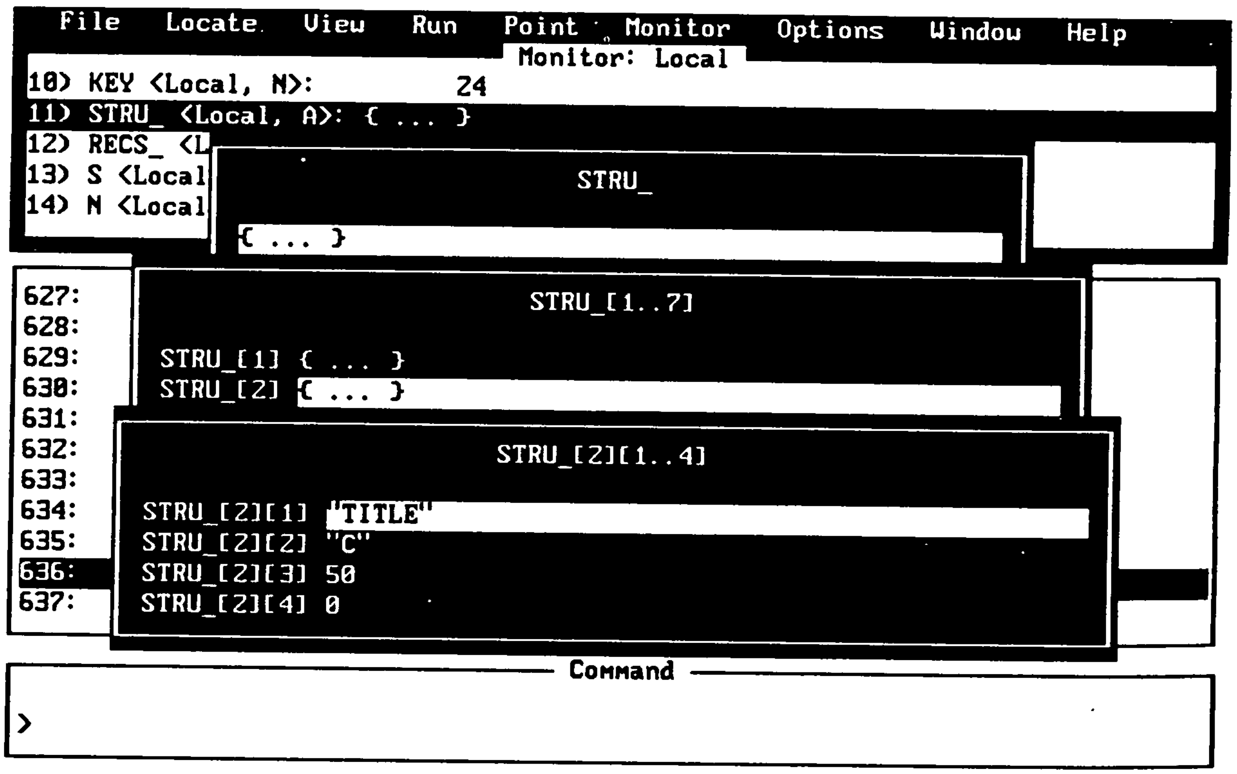
The debugger can alter the value of any variable displayed in the Watch or Monitor windows. Tab over the window, move the highlight to the desired variable, and press Enter. A dialog box will be displayed, allowing you to edit the value.

Contents of arrays

Unlike its Summer '87 predecessor, the Clipper 5 debugger can handle arrays, even those with multiple dimensions. To view and/or edit the contents of an array you must place it in either the Watch or Monitor window. Tab over to the window, move the highlight to the desired array, and press Enter. The dialog box indicates you are dealing with an array by displaying "{...}". If you type over the top of this value you replace the array with that value (be careful!). Press Enter on the {...}, and what happens next depends on the number of dimensions of the array. You may be presented with simple element values, in which case you can move the cursor to any element and edit the value. You may be presented with nested arrays, indicated as before with {...}, in which case you can highlight one and press Enter to see what it contains or type over the top of it and lose the underlying array. This process can be repeated until you run out of array dimensions. Press Esc to exit the array inspection window.

This feature is a great way to get a handle on arrays with multiple dimensions. You can walk your way deep into nested dimensions with the Enter key, and walk back out again with the Esc key.

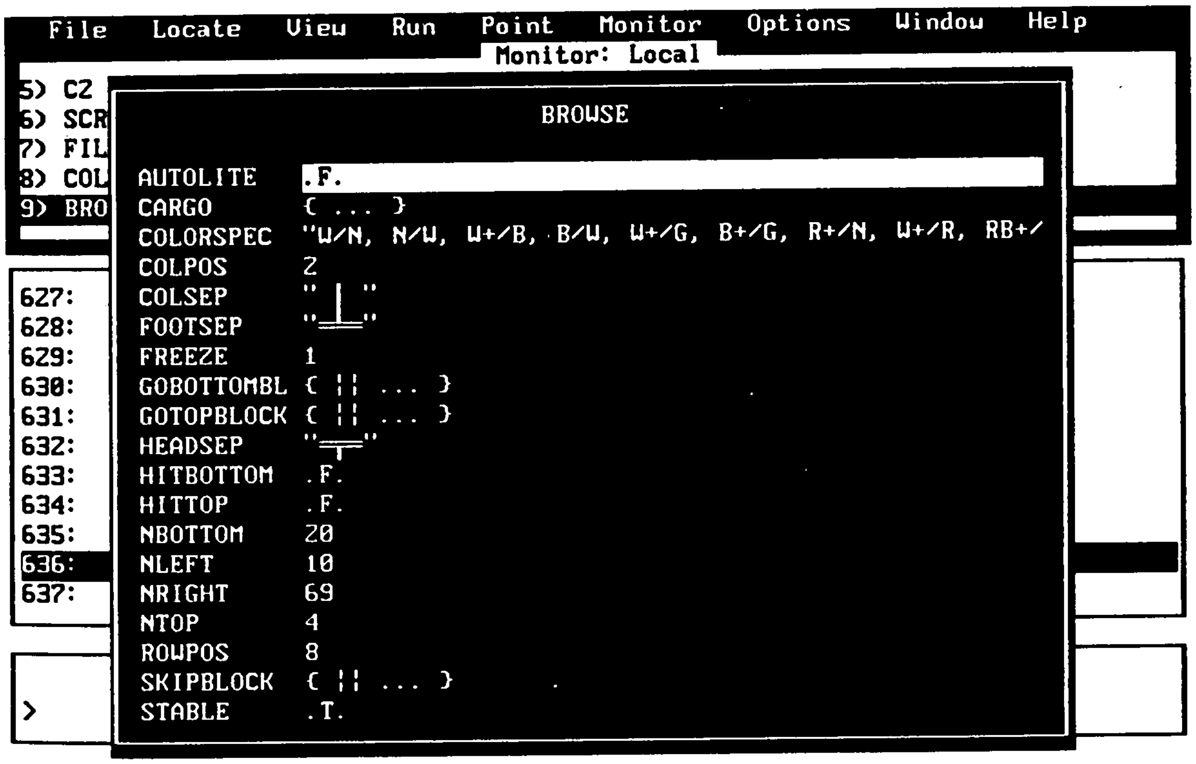
Figure 10.2 Inspecting multi-dimensional array


```
File  Locate  View  Run  Point  Monitor  Options  Window  Help
                         Monitor: Local
10) KEY <Local, N>:          24
11) STRU_ <Local, A>: { ... }
12) RECS_ <L
13) S <Local
14) N <Local
                STRU_
  { ... }
627:
628:            STRU_[1..7]
629:   STRU_[1] { ... }
630:   STRU_[2] { ... }
631:
632:            STRU_[2][1..4]
633:
634:   STRU_[2][1] "TITLE"
635:   STRU_[2][2] "C"
636:   STRU_[2][3] 50
637:   STRU_[2][4] 0
                    Command
>
```

Object instance variables

An extremely handy feature is the ability to view and alter the values of object instance variables. Similar to all the other types of variables, once placed in either a Watch or Monitor window you can press Enter when an object-variable is highlighted. Similar to arrays, the dialog box displays "Object" to indicate that the variable is an object. You can type over the top of it and lose the underlying object, or press Enter. Any of the object's instance variables that can be edited will be displayed with their current values. You can move the cursor to any instance and alter the value. Useful not only for debugging, this feature is also invaluable for learning about programming with objects in general. Press Esc to exit the object inspection window.

Figure 10.3 Inspecting object instance variables



```
5) C2
6) SCR
7) FIL
8) COL
9) BRO

627:
628:
629:
630:
631:
632:
633:
634:
635:
636:
637:

>
```

```
                         BROWSE

AUTOLITE     .F.
CARGO        { ... }
COLORSPEC    "W/N, N/W, W+/B, B/W, W+/G, B+/G, R+/N, W+/R, RB+/
COLPOS       2
COLSEP       " │ "
FOOTSEP      "═╧═"
FREEZE       1
GOBOTTOMBL   { || ... }
GOTOPBLOCK   { || ... }
HEADSEP      "═╤═"
HITBOTTOM    .F.
HITTOP       .F.
NBOTTOM      20
NLEFT        10
NRIGHT       69
NTOP         4
ROWPOS       8
SKIPBLOCK    { || ... }
STABLE       .T.
```

Global SET values

Clipper is controlled, perhaps to too great a degree, by dozens of application-wide settings. These include commands such as SET DELETED, SET SOFTSEEK, SET DECIMALS and so on. The current value of any of these settings can be viewed and even altered via the View Menu, Sets option. The effect of a change will occur immediately when the next source code line is executed. Press Esc to exit the SET inspection window.

Databases and work areas

Everything you would ever want to know about databases and other work area-specific information is available via the View menu's Workareas option (or, press the F6 key). Each database currently open is displayed in a list, and as you move the cursor through the database file names detailed information associated with the highlighted database is displayed. There's sometimes more than can comfortably fit in a single screen, so the information can be "collapsed" into an outline format. Use the Tab and Shift-Tab keys to move between sections of the display. Press Enter to collapse or expand the different screen sections in the status area.

The Callstack

The Callstack keeps track of the sequence of events that got your program to the current line of execution. If your Main() function called One() and One() called Two(), the callstack would look like the following:

```
two
one
main
```

You can Tab over to the Callstack window and move the highlight to any of the entries. Press Enter, and the program window is switched to the last line that was executed in that function. If you selected the One() function, above, the highlight will be sitting on the source code line that called function Two().

Function names prefixed with a "(b)" indicate that a code block is being evaluated within the context of that function.

Halting execution

While your application is running and in control there are several ways to interrupt it and return control to the debugger.

Crash!

For the sake of completeness we'll mention that when a fatal bug causes a run-time error, it's a sure way to get the debugger's attention. Your application will halt on the offending line. You can use the myriad debugger resources to determine what caused the error. Remember that the RESTART command (or select Restart from the Run menu) can be used to restart your application so testing and debugging can begin again without exiting all the way to DOS.

The ALT-D keystroke

As discussed previously, you can always press ALT-D to halt the application and display the debugger screen. The source code highlight will be on the line that would have been executed next.

The ALTD() function

You can invoke the debugger directly from within your application by calling the ALTD() function. This is equivalent to pressing ALT-D. The ALTD() function is very handy when used in conjunction with the preprocessor (see Chapter 7 for more details on the preprocessor). The following code fragment illustrates the use of the ALTD() function to trap an error condition that isn't fatal, but is serious nonetheless.

```
//  Place at top of program, or "compile it in" with
//  the /D compiler switch.
//
#define DEBUG

select vendor
seek vend_code
```

```
#ifdef DEBUG
  // This is a serious problem - the vendor code is
  // supposed to be found in the list. Call the debugger!
  if ! vendor->(found())
     altd()
  endif
#endif
```

Such a construction allows you to leave the debugging tests and special code in the application, permanently. When you're finished testing you can compile the source code without DEBUG being defined, and the debugging code will not be included in the resulting object code. It will still be in the source code should you need to test again. See Chapter 27, "The Error Class," for a discussion on the concept of "assertions", an additional way to trap errors of this nature. Assertions are very powerful when combined with the ALTD() function.

The ALTD() function accepts a parameter that affects both how subsequent calls to ALTD(), as well as all other debugger operations are handled. ALTD(0) disables the debugger *completely* until an ALTD(1) is encountered. While disabled, the debugger can't be invoked via the ALT-D keystroke or the ALTD() function call (with no parameter), nor are any breakpoints or tracepoints respected. ALTD(0) shuts the debugger down until the next ALTD(1).

ALTD(0) is used primarily to lock end-users out of the debugger in situations where the debugger library (CLD.LIB) is linked directly into the application. The application should default to ALTD(0) and only issue an ALTD(1) from a password-protected menu or after a special series of keystrokes. Another alternative is the use of a DOS command line parameter. Only if the parameter is specified correctly is the debugger enabled.

Another possible use is in situations where you want a section of code to run without being "caught" by tracepoints. For example, a large routine could have an ALTD(0) and ALTD(1) combination that temporarily disables the debugger whenever the routine is executed. Use of the preprocessor's conditional compilation features is important for keeping this sort of code isolated (and easily identified) from the actual function code.

Breakpoints

Breakpoints are used to halt execution when the debugger hits the specified line number in a specified function. The debugger stops before executing the line, so that you can set a breakpoint right on a line that is causing problems. The breakpoint stays active until you remove it. Setting a breakpoint on top of an existing one turns it off. You can set a breakpoint via the **Breakpoint** option on the Point menu, by issuing the BP command in the command window, or by pressing F9 on the desired line.

If you aren't sure about the line numbers for a function you can omit them. The debugger will halt on the first executable line. You can then use the F8 or F10 keys to single step through the function, or scroll through the source code and use the F7 *Run to Cursor* feature to position the debugger at the desired line.

The command line is the easiest way (short of hitting F9) to set breakpoints. You don't have to specify the name of the function when setting break points within the current source code file. The following sets a breakpoint at line 170 of the current file, at the first executable line in function ShowStat(), and at line 23 in LOADER.PRG.

```
bp 170
bp showstat
bp 23 in loader
```

Each breakpoint will be listed and associated with a sequence number. Breakpoints can be disabled by deleting them via the *Delete* option on the Point menu, or via DELETE at the command line. In either case you refer to the breakpoint by number. The following deletes the breakpoint number 2.

```
delete bp 2
```

To get rid of all breakpoints quickly you can specify ALL in the command line.

```
delete all bp
```

Tracepoints

Tracepoints are a combination watchpoint and breakpoint. You set a tracepoint on an expression just like you do with a watchpoint. The difference is that a tracepoint will halt program execution when the value of the expression changes. For example, you could set a tracepoint on a variable name. If the value of the variable changes the program will halt. In the following example, program execution will halt when the value of the **loop_count** variable changes.

```
tp loop_count
```

Tracepoints can be set on any valid Clipper expression. Keep in mind that it's not the result that matters, but a *change* in the result. Here's an amusing example. The following tracepoint will halt program execution within the next minute.

```
tp left(time(),5)
```

Note, however, that tracepoints are not evaluated during wait states in your applications. The above example will not halt execution, for example, during an INKEY(0), but the application will be halted on the very next line when the INKEY(0) is exited.

Keep in mind that tracepoints will often cause execution speed to slow down. If you have too many tracepoints, the application may slow to a crawl. Be especially frugal with tracepoints when you're using functions requiring significant amounts of processing time. All tracepoints are evaluated after each line of code is executed. Tracepoints can be deleted the same way as breakpoints.

Script files

After going to the trouble of setting up windows, monitors, watchpoints, breakpoints, tracepoints and other debugger configuration options, you can save them all to a file for later recall. The Options menu has Save and Restore choices for this purpose. When starting from DOS you can specify that a script file be used to configure the debugger immediately. In the following example a script file called MYAPP.BUG will be used in conjunction with MYAPP.EXE.

```
cld @myapp.bug myapp
```

The debugger's RESTART command also uses a script file to record all current settings, then reloads and restarts your application while retaining the previous debugger context.

The debugger will automatically load a configuration script file called INIT.CLD if one exists. INIT.CLD is an ideal way to customize the debugger. To create such a file simply save the debugger configuration as INIT.CLD.

The command line

As mentioned briefly in previous discussions, the command line can be used to perform all the major debugger functions. It's a matter of personal preference whether you find it easier to type commands or make menu selections. We find ourselves doing both. Some operations, like setting watchpoints and breakpoints, are more efficient when done from the command line. Others, like specifying which class of variable to monitor, require fewer keystrokes if selected from the menu. Table 10.2 lists the available commands.

Table 10.2 Alphabetic list of debugger commands

?	Display the value of a variable or expression
animate	Run application in animation mode
bp	Set a breakpoint
callstack	Display the callstack window
delete	Delete one, some or all "point" settings
DOS	Drop to DOS without exiting current application
find	Search for a character string in current file
go	Run the application
goto	Move cursor to specified line within file
help	Display the help screen
input	Read debugger commands from specified file
list	List some or all "point" settings
next	Search for the next occurrence of a character string
num	Toggle the display of line numbers in the code window
output	Display application screen
prev	Search for the previous occurrence of a character string
quit	Exit the debugger
restart	Reload and restart application but keep debugger settings
resume	Return to the debugger after viewing a file
speed	Specify the animation speed
step	Execute current source code line and stop
tp	Set a tracepoint
view	View the specified file
wp	Set a watchpoint

The Nantucket documentation goes into considerable detail about the use of these commands. Remember that many of them have function key short-cuts, like F8 for STEP, and that all can be accessed from the debugger's menu.

The command line keeps a history of your entries. You can recall the most recent entry via the F3 key, or you use the Up and Down keys to scroll through previous entries, and then press Enter to execute the command.

Adding your own features

A very useful side effect of the debugger's ability to evaluate any legal Clipper expression is that you can add your own debugger features. All you need to do is make sure the functions have been linked into the current application. An easy way to do this is via the EXTERNAL command. The following example creates references to the ShowStats() and DispMem() functions, but only if the DEBUG manifest constant is currently defined (see Chapter 7, "The Preprocessor," for details).

```
#ifdef DEBUG
  external SHOWSTATS, DISPMEM, LOGGER
#endif
```

At the debugger's command line, use the ? command to cause the function to be called.

```
? ShowStats()
? Dispmem()
? Logger()
```

From inside the function you can do whatever you want. All screen activity is routed to the debugger's screen and not your application screen.

For example, Listing 10.1 shows how you can pop up a box showing memory status.

Listing 10.1 Display memory status in a pop-up box

```
function DispMem(n)
/*
   Display available memory stats.
   Return .T. if it falls below specified level (optional).
*/
local clr
if n == nil
   clr := setcolor("W+/R")
   @ 10, 40 to 16, 61 double
   @ 11, 41 say "Memory Status     KB"
   @ 12, 41 say "────────────────────"
   @ 13, 41 say "Total available " +str(memory(0),   4)
   @ 14, 41 say "Largest block   " +str(memory(1),   4)
   @ 15, 41 say "RUN area        " +str(memory(3),   4)
   inkey(0)
   setcolor(clr)
   n := 0
endif
return (memory(0) < n)
```

Note that the function returns .t. if memory falls below the optional specified value. This allows us to use the function as a tracepoint. In the following example the function will cause the application to halt when available memory falls below 125 KB.

```
tp DispMem(125)
```

To pop up the status box, use the ? command and DispMem() with no parameters.

```
? dispmem()
```

Other uses for such functions include writing key information to a log file while the application is running (a form of profiling) and displaying the status of things that are too complex to monitor via a direct tracepoint or watchpoint expression. Use your imagination! If you can link it into your Clipper application you can call it from the debugger.

Classic combinations

There are many combinations of debugger features and functions that may not immediately come to mind. In the final section of this chapter we present a few "classic combinations" of features that seem to be made for each other.

Callstack and Monitor

Combining the Callstack and Monitor windows allows you to browse through the variables in all pending functions in the calling sequence. For example, monitor **locals** and **statics** and then turn on the Callstack window. Select the Callstack window by pressing tab until the window is highlighted. As you move the cursor up and down through the callstack, note how the values of the monitored variables change. If a function has no variables of the monitored class, none are displayed. This often yields an enlightening peek into the havoc caused by private variables.

Monitor and block locals

Code blocks are hard enough to understand when you write them, much less when they are being executed at run-time. At first blush it appears impossible to see what's going on inside a code block while it is being evaluated. However, a combination of Callstack and Monitor windows allows you to peer inside a code block. Let's start with the following simple example. Once you understand the basic technique you can apply it to more complicated situations. (See Chapter 8, "Code Blocks," for more details — you must understand code blocks before you try to debug them.)

First, enter the following into a single source code file. Compile (with /B!), link, and run the resulting .EXE through the debugger. We don't care about anything being displayed on the application screen, so don't run the debugger in /S (split screen) mode. A full screen view of the debugger is easier to use.

```
function Main()
local b
b := { |x,y,z| ;
        BlockHalt(1), ;
        x := 10, y := 20, z := 30, ;
        BlockHalt(2) ;
       }
eval(b, "A", "B")
return nil

function BlockHalt(v)
return v
```

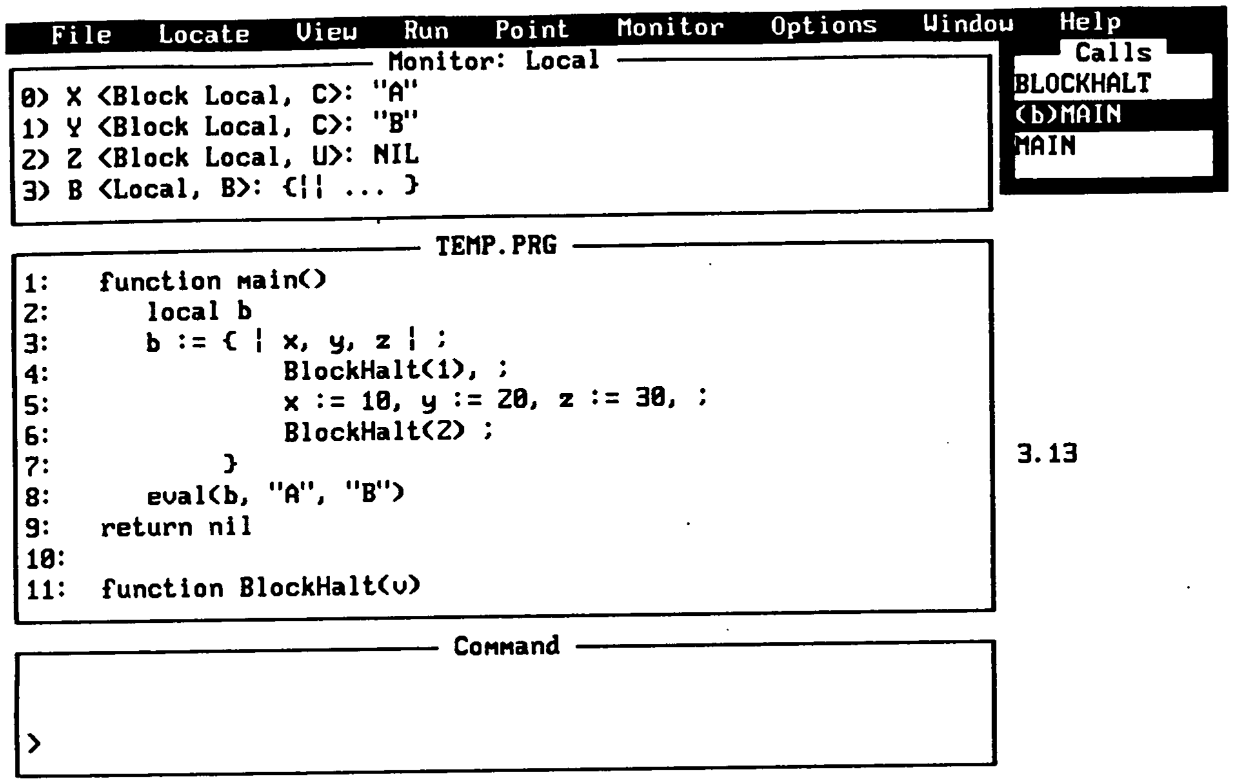
Before executing any lines of code:

1. Select Monitor:Local
2. Select View:Callstack
3. Set a breakpoint on the "return v" line in function BlockHalt(), and
4. Use the Tab key to select the Callstack window

When this is all set up, press the F5 key to begin execution. The breakpoint will cause a halt at the first call to BlockHalt() inside the code block. Since you have the Callstack window selected, move the highlight to the (b)Main line. The (b) indicates a code block is being evaluated in that function. While the (b)Main line is highlighted, the Monitor window will display any local variables that are in scope.

Within the code block, variables named **x**, **y**, and **z** are in scope. The display will clearly demonstrate the relationship between the parameters specified in the EVAL() function and the parameters accepted in the code block.

Figure 10.4 Inspecting callstack and local block variables



```
Monitor: Local
0) X <Block Local, C>: "A"
1) Y <Block Local, C>: "B"
2) Z <Block Local, U>: NIL
3) B <Local, B>: {|| ... }
```

```
Calls
BLOCKHALT
(b)MAIN
MAIN
```

```
TEMP.PRG
function main()
   local b
   b := { | x, y, z | ;
            BlockHalt(1), ;
            x := 10, y := 20, z := 30, ;
            BlockHalt(2) ;
        }
   eval(b, "A", "B")
return nil

function BlockHalt(v)
```

```
Command
>
```

```
x  "A"
y  "B"
z  nil
```

The breakpoint halted execution at the first place BlockHalt() is called within the code block, so the parameter values are displayed as they exist at the point the code block is initially evaluated. Since we didn't pass a third parameter, the value is NIL.

Press the F5 key to resume execution of the application. The breakpoint in BlockHalt() is still in effect, so the application will halt on the second call to BlockHalt() made from within the code block. Once again, move the Callstack window highlight to the (b)Main line. Note how the Monitor window reflects the local variables that are in scope with respect to the code block. The values have been changed and **z** is no longer NIL, due to the assignments made inside the code block.

You can safely add calls to BlockHalt() anywhere in any code block, since all it does is return the value passed to it. In the following example BlockHalt() will have no net effect on the code block — **num** is multiplied by 10.

```
b := { |num| BlockHalt(num) *10 }
```

In the previous example we numbered each call to BlockHalt(), making it easy to determine which occurrence of the function call is being trapped (the value of **v**, being local to BlockHalt(), is displayed in the Monitor window). Used this way, the last call to BlockHalt() will have a net effect on the code block, since the return value of BlockHalt() will be passed back to the call to EVAL().

You can even preprocess it right out of existence if you want to eliminate the wasted function call when trying for maximum run-time speed.

```
#ifndef DEBUG
  #translate BlockHalt(<v>) => <v>
#endif
```

Watchpoints on database info

There's nothing special about memory variables with respect to watchpoints. The contents of databases and the functions related to their status work just as well. You can set watchpoints on field names as well as RECCOUNT(), RECNO(), EOF(), BOF() and the rest. Use the alias operator to zero in on specific databases. The following commands establish watchpoints on the vendor database's EOF() flag and the value of the Vend_ID field in the invoice database.

```
wp vendor->(eof())
wp invoice->Vend_ID
```

Tracepoints on database info

Are you starting to get the picture? Anything that can be watched can just as easily be *traced.* Why sit and stare at the watchpoint window, waiting for the vendor database to hit end-of-file when you can have the computer do it? The following examples will cause execution to halt when either the vendor database hits end-of-file or a seek fails in the invoice file.

```
tp vendor->(eof())
tp ! invoice->(found())
```

Summary

Now that you know what the Clipper debugger needs for most effective use, and have been introduced to its basic operation, you can take full advantage of the powerful features it offers. You've also been clued in on some important tips and techniques to tap into easily overlooked capabilities. May your bugs be few and far between. Happy hunting!

Reexamination 90/005,727

The Original

Page 406 of Part II

Is Missing

Initial : AR
Date : 6-8-2000

CHAPTER 11

Designing Database Files

The first step in designing a computer application is to determine the data that needs to be processed and to organize this data in a meaningful way. A telephone book contains a lot of information, but if it were not organized, it wouldn't be much use. Imagine trying to look up a phone number if the phone book was in a random order.

In this chapter we will discuss how to determine the data an application needs and how to organize this data into a coherent file system using the .DBF file structure. We will also discuss data keys and the relationships among files. Finally, we will look at how Clipper can be used to create .DBF files once you've determined their structures.

There are several steps involved in designing a file system from a list of data items. These steps are:

- Systems overview
- Organize entities into logical files
- Define physical file structure
- Test your file structure

It is important during the data design phase to have interaction with and feedback from the ultimate end-users of the system. Each user will have their own view and perception of the data he or she needs to work with. The database design needs to incorporate all these views into a coherent logical structure that can be successfully modeled by the application language.

System overview

The first step in properly designing your databases is to get an overview of the system's functionality. What does the current system do? What are the goals of the new system? Should the computer provide all the functionality of the current manual system?

To get an overview of the system, you should examine the current system (if one exists). The current system might consist of a series of manual procedures or may be a computer system that the company has outgrown. It might be a combination of both. You should talk with all users about the system. Try to determine the entities and what information flows among them in the system.

Keep in mind that the computer will represent a model of the system. Your users will expect to be able to rely on the computer for information, and not on the manual procedures any more. Stock on hand will be determined by consulting the computer, not counting actual inventory.

Determining the data needs

The first step in determining the data that an application must maintain is to collect all the output that the system must be capable of producing. Output consists of printed reports, screen displays, information sent over modems, data transferred to other applications, etc.

The collection of output should not be limited to just the current system. You should strive to include anticipated future needs and changes. It is important to realize that a database is a constantly changing entity. What was extra information in last year's databases may suddenly be required to comply with new tax laws this year.

The future data needs can best be discovered by "blue-skying" with the end-users. Blue-skying means asking the users what they would like to see if there were no limits on the computer or their budgets. For example, a clerk using the A/R system might feel a wonderful feature would be a notes field to record comments while he made collection phone calls. Of course, since the clerk does not know the .DBF structure, this feature may seem like a luxury. Imagine his surprise when your finished design provides that note field.

Identify entities

The system overview will help you identify the entities that your system must work with. An entity is a person or object that the system uses in some fashion. For a payroll system, the entities would include the employee, the paycheck, the time card, etc.

Assume each piece of paper the system must track is an entity and that each organization/department/position is also an entity. Ask whether the system must track information about the entity and if the entity can be uniquely identified. If the answer to both questions is yes, the item probably should be considered an entity (which will eventually become a file) in the system.

All data in the system will belong with one entity or another. Each item in your data dictionary will belong to one file or another when the application is designed for the computer.

Determine relationships

Each entity in the system may be related to other entities. An entity is related to another entity if it owns the other entity, is created by it, etc. For example, an invoice entity belongs to a particular customer. A part might be supplied by one or more suppliers.

Each relationship must be classified as one-to-one, one-to- many, or many-to-many. A one-to-one relationship means that an entity can be related to only one other entity. These are most frequently fields in a physical file.

A one-to-many relationship means that one entity can have multiple occurrences of the second entity, but the second entity can have only one occurrence of the first. Our customer-to-invoice example is a one-to-many relationship. A customer can have many invoices, but each invoice has only one customer associated with it.

A many-to-many relationship occurs when each entity can be related to more than one record in the other entity. For example, a purchasing system which has many sources for parts contains a many-to-many relationship between vendors and parts. Each part can be supplied by many vendors and each vendor supplies many parts.

It is important during this step to classify each relationship. The design of the files will be contingent upon the types of relationships.

Obtain copies of all output the system must produce

During your system review phase, you should obtain copies of all output that the system must produce. This should include planned screen displays and printed reports. It should not, however, be limited just to visible displays/reports only. If the system must produce data to transfer to another system, this data is considered part of the output as well.

The collection of all output will be used to test your file structure later on. Since your computer system needs to produce all of the output, each item in the output must be found somewhere in the system.

Build a data dictionary

Once you've determined the data needed both for the current system and expected future use, you should organize the data into some kind of data dictionary. A data dictionary is a list of all information in the system, along with brief descriptions of its purpose and possibly information about the data type and size. Figure 11.1 illustrates a sample format for a simple data dictionary file stored in a .DBF structure.

Figure 11.1 Sample data dictionary structure

No.	*Field*	*Type*	*Size*	*Description*
1	ELEMENT	C	10	Name of this data item
2	DESC	C	50	Brief description of item
3	SIZE	N	6,2	Size of data item
4	TYPE	C	10	General classification of data
5	FILE	C	8	File name where item belongs
6	DEFAULT	C	25	Default value
7	REQUIRED	L	1	Is field required?
8	IS_KEY	L	1	Is this item a key field?
9	SOURCE	C	1	(S)torage or (D)erived

Currently, much work is being done by various xBASE vendors to define a standard data dictionary for .DBF files. The current .DBF structure is limited to name, type, and size. The future dictionary might include validations, help prompts, required flags, and so on. It is hoped that the xBASE vendors will cooperate so a common standard dictionary will result, instead of many different versions.

It is important while preparing the data dictionary to think in terms of the future. For example, one application tracked the movement of new cars to respective dealerships. A single digit was used for the number of cars that could be carried on a truck. As cars got smaller and carrier trucks got larger, it became possible to carry ten cars on a single truck. Due to the file design, many programs had to be rewritten so that X could represent 10 cars and Y 11 cars.

Once the data dictionary is created, each item needs to be reviewed to determine how the item is derived. There are five general sources from which a data item may be obtained.

Constant or hard-coded data

Some data might be hard-coded into the application program. A good example would be the prompts that appear on a menu. (You did recognize menu prompts as a screen display of output data, didn't you?) If the application is not expected ever to have different prompts, these might be candidates for data that you hard-code into the application.

Any data that's hard-coded into the program will require the program to be changed and recompiled if the data ever needs to change. Making the change might be time-consuming, but such data is very secure from the end-users, unless they happen to be programmers.

Carefully consider any data before you hard-code it. Even the menu prompts might have to be changed if the system is going to be distributed in another country. While hard-coding data also has the advantage of being easy to create, it does require a programmer to change it.

Derived data

Data can also be derived from calculations. A good example would be the total price of several items ordered on an invoice. Since this data can be readily derived from the number of items sold and the price per item, it should not be stored separately in a .DBF file.

In general, a calculation is based on other information in the system and some sort of operation. Derived data is any data which can be created by applying some sort of operation to existing data. For example, in the data dictionary in Figure 11.2, the sales tax is derived from the sales tax rate times the total amount due on the invoice.

Figure 11.2 Derived sample data dictionary

Element	*Desccription*	*Size*	*Type*	*Source*
AMOUNT	Total amount of invoice	11,2	Numeric	(S)torage
TAX_RATE	Sales tax rate	5,2	Numeric	(S)torage
SALES_TAX	Derived from AMOUNT * TAX_RATE	11,2	Numeric	(D)erived

System data

The operating system and the DOS environment maintain various pieces of information that are available to the programmer. For example:

```
? date()          // Returns the system date
? time()          // Returns the system time
```

If data can be extracted from the system, it should be treated as system data. It does not need to be stored, but can be called upon using some sort of function call.

The definition of system data should not be limited to the built-in functions provided by Clipper. For example, if you create a function that returns the full name of a state from a two-character state code, the state name from your function could be viewed as system data.

Input data

Some data will also be input and does not need to be saved. For example, a user-entered password, or the temporary holding variable for the response to the menu prompt. Input data of this sort needs to be identified, just as output data does. The input data might be necessary to control how the output is produced. While the data is not saved, it still needs to be validated and is subject to type and size considerations.

views of the file. Any data which the system will need to access repeatedly should be stored in the file structures. For example, in an accounts receivable system, the customer's billing address and phone number would be stored in the customer file.

After all the data is classified, a subset of the data dictionary should be created with just the stored fields. This subset will become the basis for our logical file views, discussed in the next session.

Eliminating redundant data

Once the list of stored fields has been created, we need to review it for redundant information. Redundant information is data that is being maintained in more than one spot. If the same information is updated in multiple programs, it is very difficult to maintain data consistency and to ensure the integrity of the information.

For example, a company might have its payroll and personnel systems computerized. As information is entered into both applications, the possibility exists that an end-user may look at a printed employee report and see different information than his or her terminal displays on that report. If such a situation arises, needless to say, it does not inspire much confidence in the computer system.

To eliminate redundancy, we must first organize our dictionary by data item. As we do this, we need to carefully review each piece of data to see if it truly is a duplicate of existing information. For example, let's assume we are computerizing an existing accounts receivable system. This system tracks customers, invoices, and payments. During our data determination step, we discover that the customer information is kept in rolodexes by the salesmen and in a box of index cards by the invoicing department. A segment of our data dictionary is shown in Figure 11.3.

Figure 11.3 Subset of Data Dictionary

Element	*Description*	*Size*	*Type*
RADDRESS	Billing address in rolodex	25	Character
IADDRESS	Billing address on index cards	25	Character
RCITY	City in rolodex	15	Character
ICITY	City on index cards	15	Character
RCOMPANY	Company name from rolodex	30	Character
ICOMPANY	Company name on index cards	30	Character
RCONTACT	Contact name from rolodex	25	Character
ICONTACT	Contact name on index cards	25	Character

At first perusal of Figure 11.3, it appears that all the data is redundant. This may very well be the case. However, upon talking to the end-users, we discover that the address, city, and company name are truly redundant, but the salesman's contact is normally the purchasing agent and the invoicing department's is usually the bookkeeper or accounts payable clerk. After we remove the redundancy, Figure 11.4 illustrates our new data dictionary.

Figure 11.4 Data dictionary with redundant data removed

Element	*Description*	*Size*	*Type*
ADDRESS	Customer billing address	25	Character
CITY	Billing city	15	Character
COMPANY	Company name	30	Character
PURCHASING	Name of purchasing agent	25	Character
BOOKKEEPER	Name of bookkeeper	25	Character

The process of removing redundant data should be done with the help of the end-users. Data that appears to be redundant on our first pass through the dictionary might not really be redundant. The dictionary should be reviewed as redundant data is removed to ensure that all output that will ever be needed can still be produced.

Redundant data can sometimes be left in the file if there is a good reason for doing so. For example, suppose in our A/R system the bookkeeping department wants an aging list of all customers with their balances due. Figure 11.5 illustrates a sample aging report.

Figure 11.5 Sample aging report

Customer	*Current Balance*	*31-60 days*	*61-90 days*	*over 90*
ASHTON-TATE	1,000.00	500.00	250.00	100.00
NANTUCKET	600.00	1,200.00	0.00	50.00
FOX	100.00	0.00	0.00	20.00

As we reviewed the data we determined that the balance field and the aging fields (31-60 days, 61-90 days, and over 90 days) are all derived fields calculated by totalling the invoices based on invoice date.

This means that whenever the aging report needs to be run, the system needs to calculate four field values for each customer. Since this will require that the invoice file be totalled each time, performance of the report could slow down for customers with many invoices.

From a theoretical standpoint, the data should be derived to eliminate the redundancy; however, this might not be practical. If the company has thousands of customers and hundreds of thousands of invoices, the bookkeeper requesting the report might kick off a computer process of two hours or more.

If we accept some redundancy in the file, we could create a separate function called Aging() which would calculate the age fields and store them in the customer file. This function might be run off-hours since it will take some time to run. Then the aging report would simply need to read stored values from the database. Obviously, this report will run much more rapidly.

Even though the report now runs more rapidly, it is only as accurate as the last aging run. The date of the last aging run should be included in the report header. This way, when a clerk displays a customer balance on a terminal that appears different on the report, they can look at the date and realize that the customer has had transactions processed since the aging report was last run.

Redundancy should be carefully considered before it is removed. First, you must make sure that the data is truly redundant, not just different data with the same element name. Second, you must weigh the performance considerations if all redundancy is removed. These should be carefully considered against the extra programming effort required in allowing redundant data.

Organize entities into files

Once you have completed your overview, you should have identified the logical entities with which the system must work. For the most part, each logical entity will represent a logical file in your finished structure. These logical files may require more than one physical .DBF file. The logical file is the end-user's view of an entity within the system. The physical file is the actual .DBF where the data is stored.

For example, a company might maintain a logical file of customers. Each customer has a bill-to address and one or more shipping addresses. To the end-user, this is one logical file, called the "CUSTOMER" file. Yet, when we write the program to work with this file, we would probably create two physical .DBF files, one to hold each customer's billing information and a second to hold the customer's shipping locations. Data normalization will help organize logical files into physical files.

Assign primary and alternate keys

Each logical file in the system should have at least one key. This key must uniquely identify the record and distinguish it from all other records in the file. This key is known as the primary key.

The primary key should be a key that has some meaning to the end-user. For an employee, a good primary key would be the social security number. It is unique and the end-user easily understands it. For other databases, a primary key might need to be created. A customer file would probably have a customer ID code created for it. These customer IDs would be unique to the customer.

Alternate keys are fields which provide alternative access into the file. These keys do not have to be unique. For example, in our customer file, the primary key is the customer ID code. A secondary key might exist for the company name, since the end-user might not remember the code but will recognize the company name when it is displayed on the screen. It is important that the end-users are aware when they use a secondary key that duplicates might exist. When displaying a list of company names, you might also display the company's city and state. This will allow the end-user to select the company they want even if two companies have the same name.

Examples of keys

In order to understand the selection of keys, let's work through an example. Assume a customer file has the structure illustrated in Figure 11.6.

Figure 11.6 Customer file structure

Field name	*Type*	*Size*	*Description*
COMPANY	Char	30	Company name
PURCHASING	Char	25	Purchasing agent
BOOKKEEPER	Char	25	Bookkeeper or A/P clerk
ADDRESS	Char	25	Billing address
CITY	Char	15	Billing city
STATE	Char	2	State
ZIPCODE	Char	10	Zip code
PHONE	Char	14	Area code and phone number
SALES_REP	Char	3	Initials of salesman
CRED_LIMIT	Num	9,2	Maximum credit limit

As we look through this list of fields, the only likely key fields are the COMPANY and PHONE fields. The company name has the potential of being duplicated and the phone number is difficult to remember. So we need to add a new field for a company ID. This field will consist of a user-assigned unique abbreviation for each company in the file. When we create a data entry screen for customers, we will prompt for a customer ID. Our program will have to make sure no duplicate customer ID codes are entered.

Now that we've defined the primary key, let's look for candidates for secondary keys. If we expect to analyze sales by territory, the STATE and ZIPCODE fields combined would probably be a good secondary key. In addition, the SALES_REP field would also likely be a secondary key. If we expect that the customer may be calling in, the phone number would be a good secondary key, since the customer knows his own phone number but probably does not know the customer ID code we've assigned as the primary key.

Assign data items to logical files

Each data item in your data dictionary should now be mapped to a particular logical file. This is done by comparing each element with the keys and determining the key with which the element is associated. For example, if our customer file has a key field of customer-id, we would expect customer name, address, billing contact, etc., to be placed into the logical customer file. The main question for each key is, "If I find this key in a database, what additional fields do I expect to find with it?" Each element in your data dictionary should end up being associated with one primary key. If your relationship shows files linked together, a foreign key will be placed in one file to refer to the primary key of the file to which it is linked.

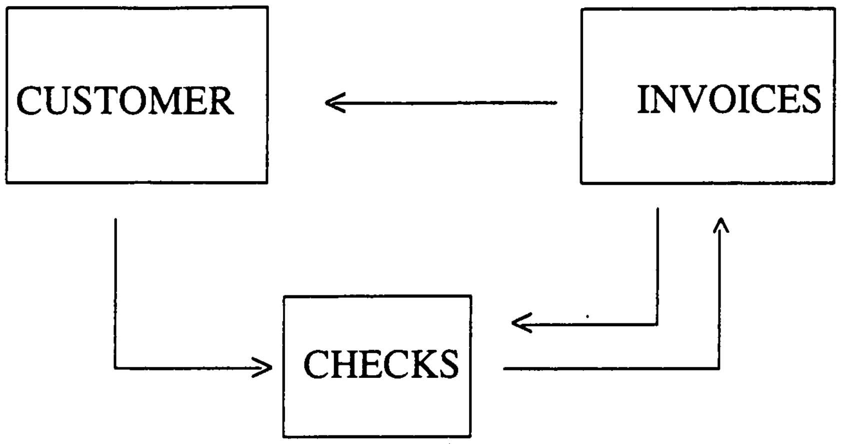
Organize logical files into physical files

The logical files you've identified up to this point need to be represented in a way the computer can efficiently work with them. What is one file to the end-user might be several files to the computer. There are two main steps in determining your physical file requirements.

Break down many-to-many relationships

Clipper and other relational database packages cannot handle many-to-many relationships. If you have entities which share a many-to-many relationship, it's necessary to break that relationship apart. A many-to-many relationship occurs when each entry in one file can be associated with several entries in another file. The second file can also be associated with several entries from the first file. For example, Figure 11.7 shows a many-to-many relationship between checks and invoices.

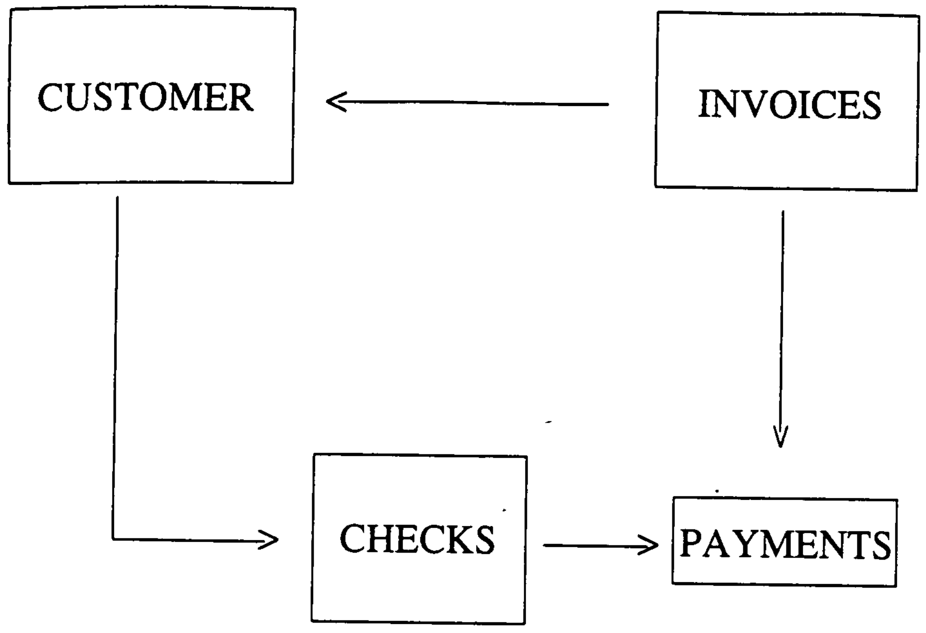
Figure 11.7 Many-to-many relationship example



The extra set of arrows connecting INVOICES and CHECKS illustrates a many-to-many relationship. Each invoice might be paid by more than one check and each check might cover more than one invoice.

Since Clipper does not support the many-to-many relationship, we have to model it ourselves. This is done by creating a new database which sits between the two logical files. Figure 11.8 shows how to use an intermediate file to convert a many-to-many relationship into two one-to-many relationships.

Figure 11.8 Broken Many-to-many Relationship



The new PAYMENTS file is used as an intermediate file to remove the many-to-many relationship between the INVOICES and CHECKS files. Now the CHECKS file has a one-to-many relationship with the check number of the PAYMENTS file. One check may have multiple payments, but each payment corresponds to only one check. Similarly, a one-to-many relationship exists between PAYMENTS and INVOICES. Each invoice may have several payments against it, but each payment corresponds to only one invoice record.

The PAYMENTS file in this example is a link file. There is no real world equivalent in the system; it is used only to allow the computer to represent a concept (many-to-many) that is permitted in the real world.

The structure of the PAYMENTS file is shown in Figure 11.9.

Figure 11.9 PAYMENTS.DBF structure

Field name	*Type*	*Size*	*Description*
CHECK_NO	Char	8	Foreign key into check file
INVOICE_NO	Char	12	Foreign key into invoice file
AMOUNT	Num	9,2	Amount of check applied to this invoice.

The PAYMENTS file would have two indexes, one on CHECK_NO and one on INVOICE_NO. These keys are known as foreign keys. A foreign key is a key in a database which is used to link to another database. The key must be the primary key in the related files.

When a check is received, it will be recorded in the check file and an entry will be made in the payment file for each invoice that the check pays. If we need a list of checks that paid a particular invoice, the PAYMENTS file will be selected and searched for all records with the INVOICE_NO number as the key.

Normalize the files

Normalization is an attempt to simplify the files and remove redundant data. Each logical file in your design should be normalized into a set of physical files that can safely model the logical file's structure.

The logical files you've derived from the data dictionary will become the basis for the actual .DBFs the system works with. Logical files represent entities that the end-user views in the system. However, these logical files might not be easily represented in a relational database. The process of data normalization can be used to convert logical files into tables which can be represented via relational data structures.

To normalize a logical file, several steps are taken to eliminate redundant data and help ensure data integrity in the system. While discussions of normalization theory can easily fill books, a brief summary of the key operations is presented here. If you need more details, refer to works by Codd and Date.

Place repeating fields into separate files

All repeating groups should be removed into a separate table or tables. For example, a logical file called INVOICE might contain a customer code, an invoice number, a sales rep, an order date, and so on. In addition, for each line item on the invoice there is a quantity, part number, and price. Figure 11.10 illustrates a logical invoice file.

Figure 11.10 Logical INVOICE file

```
CUSTOMER_ID
INVOICE_NUMBER
SALES_REP
SALESMAN_NAME
ORDER_DATE
QUANTITY     ───┐
PART_NUMBER     ├── These three fields repeat several times
PRICE        ───┘
```

We should break the INVOICE file into two files. One file will contain the heading information and the second file the information from each line item. Figure 11.11 illustrates the logical INVOICE file after the repeating fields have been moved into a separate file.

Figure 11.11 Logical INVOICE file - repeating data moved

```
Header file        Line item detail file
CUSTOMER_ID        INVOICE_NUMBER ─╲
INVOICE_NUMBER     PART_NUMBER    ─╱──── Primary key
SALES_REP          QUANTITY
SALESMAN_NAME      PRICE
ORDER_DATE
```

Notice that in breaking the structure into two files we needed to duplicate the INVOICE_NUMBER field. In the line item detail file, this is known as a foreign key. A foreign key in one table must be a primary key in another. It is used to link the files together. Each entry in the line item file would be linked to the header file via the INVOICE_NUMBER field.

Remove non-dependent fields to separate files

Once all repeating groups are removed, you should review your databases for non-dependent data. Non-dependent data is data which is not solely dependent upon the primary key of the logical file.

In our line item detail file, the PRICE is not dependent upon the primary key, but it is dependent upon the PART_NUMBER. Therefore, a second file would be created which contains the part number prices. In addition, the SALESMAN_NAME field is dependent upon the SALES_REP value, not the INVOICE_NUMBER field.

Figure 11.12 illustrates the final version of the INVOICE file. Notice that it has been broken into four separate files.

Figure 11.12 INVOICE file - final version

Header file
```
CUSTOMER_ID
INVOICE_NUMBER ——— Key
SALES_REP
ORDER_DATE
```

Sales Rep file
```
SALES_REP           Key
SALESMAN_NAME ———
```

Line item Detail file
```
INVOICE_NUMBER
PART_NUMBER      >——  Key
QUANTITY
```

Part file
```
PART_NUMBER      ———  Key
PRICE
```

Test your file structure

Once you have finished your file structure, you should test it to see if it can do everything you need it to do. If not, plan on revising the structure in a recursive manner. Make your revision, and test again until your structure is solid. The work you do in properly designing your files will more than pay for itself when you write code.

Can all output be produced from your files?

After you've done a preliminary breakdown of logical files into physical files, you should review the sample output you've acquired during the system overview. For each output item, look at each field on the screen or report. Each data item should be either system-generated, such as the date or time, calculated such as report totals, or most importantly, capable of being extracted from the database. For example, Figure 11.13 shows a sample Customer Balance Due List.

Figure 11.13 Customer Balance Due Report

```
Report Date: Jan 22, 1991      <A>
Report Time: 4:50:00 am

                  JJ & Associates Company   <B>
                  Customer Balance Due Report

Customer    Contact     Phone #       Balance Due    <C>

xxxxxxxxxx  xxxxxxxxxx  xxx-xxx-xxxx  9,999.99
xxxxxxxxxx  xxxxxxxxxx  xxx-xxx-xxxx  9,999.99
xxxxxxxxxx  xxxxxxxxxx  xxx-xxx-xxxx  9,999.99
xxxxxxxxxx  xxxxxxxxxx  xxx-xxx-xxxx  9,999.99

   <D>         <E>          <F>         <G>

                  TOTAL BALANCE DUE  9,999.99  <H>
```

The date and time <A> are probably obtained from the system functions, while the report title <B> and column headings <C> are probably hard-coded data. The actual data items, <D> through <G> are probably pulled from the customer file, although item <G> might be calculated by totalling all the invoices the customer has open. The grand total <H> is probably a sum of all open balances as the report is produced.

By reviewing each report and screen we might identify areas where data cannot easily be obtained. For example, if no balance field exists in the customer file, the balance in the report must be derived from all open invoices. Can this easily be accomplished? If so, how many invoices will a customer typically have open? If the number is high, this report could take considerable time to produce. Will that be acceptable or will the report be run so frequently that processing time is a factor?

Referential integrity

Referential integrity means that any foreign field in any database has either a NULL value or can be found as a primary field in a different database.

Referring to our INVOICE databases in Figure 11.12, we see that the SALES_REP in the INVOICE file is a foreign field to link to the SALES REP file. To be sure that the data structure is intact, the value in the SALES_REP field of the INVOICE database must be either empty or must be a value from the SALES REP file. If any other value exists, the database is corrupt.

You should carefully review your design at this point, and throughout development, to ensure that any time a foreign field is updated it is checked against the appropriate primary key in another file. Also, any time a primary key is deleted, be sure that the deletion will not create a failure of referential integrity.

Organizing the logical structure into .DBF files

Once you've defined a normalized set of files, it is necessary to convert that logical view into a physical set of files the computer can understand.

Computers can identify data only as a bit. All computers are designed to recognize a bit as ON or OFF. As Figure 11.14 illustrates, there are quite a few groupings between the bits the computer "sees" and the "logical database" that the end-user works with.

Figure 11.14 Bits to databases

A computer recognizes bits being only ON or OFF.

Bit	An on/off state, such as 1/0 or Yes/No, etc.
Byte	8 bits per byte, allows for 256 different characters.
Field	A group of related characters, such as name, address, salary, etc.
Record	A group of fields which are related together.
File	A group of related records.
Database	A system of related files.

The End-user recognizes information in a text or graphic fashion.

Clipper's data management system starts at the field level. For designing applications with Clipper, we must therefore start our physical file structure at this same level. Since Clipper will enforce certain rules and restrictions on fields that we define, we can use the built-in features of Clipper's DMS (Data Management System) to our advantage.

A file definition in Clipper consists of a list of fields. Each field has a name, a type, and a size. As we look at each logical file, we can equate that file's attributes with fields in a .DBF file. Clipper supports four field types: numeric, date, logical, and character. The character type can be fixed length or variable length (as in a memo field).

Creating .DBF files

Clipper can read and write files created by dBASE and other xBASE interpreters. It can also create .DBF structures. As each version of xBASE products comes out, there is a possibility that Clipper might not be able to directly work with its files. You should include a utility in your application that creates the files you need.

DBCREATE()

The DBCREATE() function takes a two-dimensional array structure and creates a DBF according to the array's contents. Its syntax is:

```
DBCREATE( <cFilename>,<aStructure> )
```

<cFilename> is the name of the .DBF file to be created. The .DBF extension is optional. The **<aStructure>** is the array which holds the field definitions for the file. Its structure is shown in Figure 11.15:

Figure 11.15 DBCREATE() array structure

Element	*Name*	*Manifest Constant*
1	cName	DBS_NAME
2	cType	DBS_TYPE
3	nLength	DBS_LENGTH
4	nDecimals	DBS_DECIMALS

The manifest constants for the DBCREATE() function can be found in the DBSTRUCT.CH file in the \CLIPPER5\INCLUDE directory.

Listing 11.1 illustrates a sample customer file being created using DBCREATE().

Listing 11.1 Sample DBCREATE() function call

```
local c_fld :={}

Aadd( c_fld,{ "CUSTOMER"  ,"C", 8,0})     // Customer ID
Aadd( c_fld,{ "COMPANY"   ,"C",30,0})     // Company
Aadd( c_fld,{ "PURCHASE"  ,"C",25,0})     // Purchasing
Aadd( c_fld,{ "ADDRESS"   ,"C",25,0})     // Billing address
Aadd( c_fld,{ "CITY"      ,"C",15,0})     // Billing city
Aadd( c_fld,{ "STATE"     ,"C", 2,0})     // State
Aadd( c_fld,{ "ZIPCODE"   ,"C",10,0})     // Zip code
Aadd( c_fld,{ "PHONE"     ,"C",14,0})     // Phone number
Aadd( c_fld,{ "SALES_REP" ,"C", 3,0})     // Salesman initials
Aadd( c_fld,{ "CRED_LIMIT","N", 9,2})     // Credit limit
Dbcreate( "CUSTOMER.DBF",c_fld )
return nil
```

Creating index files

An index file is used to provide rapid keyed lookup to a .DBF file. It does this by creating and maintaining a binary tree which reduces disk searching time. The INDEX ON command is used to create an index file. The SET INDEX TO command is used to associates indexes with .DBF files.

INDEX ON/DBCREATEINDEX()

Either the INDEX ON command or the DBCREATEINDEX() function can be used to create a supplemental index file. The syntax is:

```
// index on command

INDEX ON <expression> TO <cFile_name> [unique]

//  dbcreatindex() function

DBCREATEINDEX( <cFile_name>,<expression>[,<bExpr>],;
                               [lUnique] )
```

<expression> can be any valid Clipper expression, including user-defined functions. The expression must evaluate to the same length for each record in the database.

Using the ***unique*** keyword or setting **lUNIQUE** to .T. instructs Clipper to write only one entry for each occurrence of a key. If a key value returned by the index expression appears more than once, only the first occurrence will be written into the index.

SET INDEX TO/DBSETINDEX()

The SET INDEX TO command or the DBSETINDEX() function can be used to specify the list of index files that should be associated with a particular database file when it is used. It is the same as specifying an index list in the USE command. The syntax for set index is:

```
SET INDEX TO <cIndex_file(s)>
```

The syntax for DBSETINDEX() is:

```
DBSETINDEX( <cIndex_file>,<bExpression> )
```

Once the index list is set, all indexes in the list will be updated whenever a database operation is performed. If there are no index files listed, the set index command closes all open indexes in the current work area.

CHAPTER 17

Pop-up Programming

Early in the 1980s, Borland International released a product called Sidekick. This handy program contained a calculator, a notepad, and a calendar. What was unique about it was that all of these utilities could be called up while another program was running. Sidekick was one of the first commercially available pop-up programs. It could appear on the screen on top of the existing program, allow you to do your work, and then quietly disappear back into the depths of the computer's memory. The concept offered a new aspect of programming that had previously not been thought to be available on a DOS computer.

Pop-up programming is now an accepted part of application development. Users expect to have access to pop-up calendars, calculators, etc. while they are in the midst of their program. While early releases of Clipper allowed the programming of pop-ups, Clipper 5 provides very powerful tools to successfully implement pop-up programming. In this chapter we will discuss how Clipper can be used to assign pop-up programs to keystrokes and cover the items you must consider to write an effective pop-up program.

Pop-up background

In order to create a pop-up program in DOS, we need to understand a little about how DOS operates. The designers of DOS built the system for a large degree of flexibility. One of the things they did was use the first 1024 bytes of RAM memory in DOS to hold an interrupt table. This table consists of 256 entries, each containing an address in memory that should be executed if that interrupt occurs. The interrupts provide various services to application programs. Figure 17.1 shows some sample interrupts.

Figure 17.1 Sample interrupt table

Interrupt #	*Type*	*Action causing it*
0	Hardware	Divide by zero
5	Hardware	Print screen
9	Hardware	Key was struck
17	Software	Return equipment list
18	Software	Return BIOS memory size
28	Hardware	Clock tick (18.2 times per second)

When DOS detects a condition in the table, it calls the interrupt handler. The table is searched to find the interrupt number, and once found, the address in memory to execute is known. The program at this address is then executed.

To write a DOS program that can interrupt another program, we first need to choose the interrupt we want to use to invoke our pop-up program. Most often this will be interrupt nine, the keyboard interrupt.

Once we've chosen the interrupt, we need to change the original address to a new address somewhere within our program. Now when the interrupt occurs, control will pass to our application, rather than to the normal DOS program. In this way, we can keep control while another program is running.

Of course, since the interrupt table now points to an address in memory, we must have a way to keep our program in memory. DOS provides a special function called KEEP or **Terminate and Stay Resident** (TSR). This function instructs DOS to exit the program but save a portion of it in RAM. While this action allows our program to gain control during an interrupt, it also requires some of the 640K of RAM available in a DOS environment.

Memory

Since Clipper requires memory for its applications to operate, the use of TSR programs in RAM while a Clipper application is being executed is generally not a good idea. So how do we keep memory for our Clipper program while we keep the end users happy? The obvious answer is, we write the pop-up feature in Clipper, and link it into our application. Fortunately, Clipper recognizes this need and provides the tools to do so.

Clipper interrupts

Clipper allows a program to be interrupted during any wait state. A wait state is any function or command that gets keys from the keyboard. These functions are listed in Table 17.1

Table 17.1 Clipper WAIT states

ACHOICE()
DBEDIT()
MEMOEDIT()
ACCEPT
INPUT
READ
WAIT

The notable exception is the INKEY() function. Later in the chapter we will show how INKEY() can easily be replaced with a function that waits for keys and still processes interrupts.

Clipper has no direct provisions for providing interrupts other than at keystrokes. However, ACHOICE(), MEMOEDIT(), DBEDIT(), and the GET SYSTEM all provide the programmer with control during the keyboard input process. Code can be written in any of these functions to perform interrupts based upon other factors, such as time.

Clipper pop-ups

Clipper provides the ability for any keystroke to call a user-defined procedure. It accomplishes this by assigning a 32 element table which holds the keys and procedures to be interrupted. This table is referred to as the SET KEY table. When a key is pressed, it is checked against the SET KEY table to determine whether or not a procedure should be called.

Keystrokes are stored in the table as numeric values. The value may be either the ASCII code for the key or a number which corresponds to their INKEY codes. The ASCII codes and a Table of INKEY values are listed in Appendix A. The inkey codes are defined in the INKEY.CH header file found in the \CLIPPER5\INCLUDE subdirectory.

Assigning keys to the SET KEY table

The SET KEY table is automatically created when a Clipper application executes. Clipper provides three commands to place keys and procedures into the table.

SET KEY command

The SET KEY command is used to place a key number and a procedure name into the set key table. Its syntax is:

```
SET KEY <nInkey_code>  TO <cProcedure_name>
```

The **<nInkey_code>** is a numeric value that corresponds to either an ASCII code or one of the keys from the INKEY table. This defines the keystroke that should cause the named procedure to be executed.

The **<cProcedure_name>** is the name of the procedure or function that should be called when the key is pressed. When the function is called, it will be passed three parameters: the procedure name that called it, the current line number (if the code was compiled with line numbers), and the name of the variable awaiting input. The procedure does not have to make use of the parameters. If you do not expect to use the parameters, you do not need to declare them in a parameter statement. For example,

```
SET KEY K_F2 TO Pop_Calc
```

will cause the Pop_Calc procedure to be called whenever the user presses the F2 function key.

An entry for the F1 function key is placed into the key table automatically when the program starts. The procedure name is HELP. You can override this by explicitly assigning another procedure name to the F1 key. For example,

```
SET KEY K_F1 TO No_Help
```

will set the F1 key to a procedure called No_Help.

SET FUNCTION command

The SET FUNCTION command is a special variation of the SET KEY command. The syntax for set function is:

```
SET FUNCTION <nFunctionKey> TO <cString>
```

The **<nFunctionKey>** is the number which refers to the function key you wish to set. Table 17.2 lists the numeric codes for the function keys. Notice that these codes are not the same as the INKEY codes from Appendix A.

Table 17.2 SET FUNCTION Key Number

Key	*Alone*	*Shift+*	*Ctrl+*	*Alt+*
F1	1	11	21	31
F2	2	12	22	32
F3	3	13	23	33
F4	4	14	24	34
F5	5	15	25	35
F6	6	16	26	36
F7	7	17	27	37
F8	8	18	28	38
F9	9	19	29	39
F10	10	20	30	40

<cString> is the character string that should be placed into the keyboard buffer whenever the function key is pressed. This character string may contain control codes such as K_LEFT to allow editing while in a wait state. For example,

```
SET FUNCTION 2 TO chr(K_CTRL_Y) + "Pennsylvania"
```

will cause the the GET to be cleared and the word "Pennsylvania" to be placed into the keyboard buffer whenever the F2 key is pressed.

The SET FUNCTION command causes a code block to be written into the set key table for the particular function key. For example, the command:

```
SET FUNCTION 2 TO "Pennsylvania"+chr(K_ENTER)
```

would result in a code block, which when evaluated will perform the following action:

```
Keyboard "Pennsylvania"+chr(13)
```

Since Clipper 5 maintains only one SET KEY table, the SET FUNCTION command will overwrite any prior definition for the key. This is a change from earlier versions of Clipper which kept the SET FUNCTION table and the SET KEY table separate.

SETKEY() function

The SETKEY() function is used both to query the SET KEY table and to update entries in the table. Both the SET KEY command and the SET FUNCTION command get translated by the preprocessor into SETKEY() function calls. The syntax for the SETKEY() function is:

```
<bCodeblock>  := SETKEY( <nInkey_code>,<bNewCode> )
```

<bCodeblock> is the block of code that is assigned to the requested key code. If no code block has been assigned to this key, the function will return a NIL. The example shows a simple UDF to determine whether a function key has been assigned or not.

```
* Program...: Is_assigned()
*
function Is_assigned( nKey )
return ( setkey(nKey)==nil)
```

<nInkey_code> is either an ASCII code or an INKEY code. It indicates the key number that should be looked up in the SET KEY table.

<bNewCode> is an optional code block to assign to the selected key code. If <bNewCode> is not specified, the set key function merely returns the current code block assigned to the key or a NIL. If a code block is specified, the current code block for the specified key will be overwritten with the new code block.

If a code block is specified, it is called whenever the designated key is pressed. The code block is passed three parameters: the procedure name, the line number, and the current variable name. The code block is then evaluated using the EVAL() function. Code blocks are discussed in greater detail in Chapter 8, "Code Blocks."

The example below assigns a pop-up calendar, calculator, and message pad to function keys two through four. These pop-up programs do not need the parameter. Function key F1 gets assigned to the User_Help() function, which needs all three parameters.

```
* Program...: SETKEY()
*
#include "INKEY.CH"
setkey( K_F1, { |cProc,nLine,cVar|User_Help(cProc,nLine,cVar) })
setkey( K_F2, { || Pop_calc() } )
setkey( K_F3, { || Pop_calend() } )
setkey( K_F4, { || Pop_while() } )
```

Changing an entry in the SET KEY table

If an entry already exists in the SET KEY table for a particular key, that entry can be overwritten by calling the SET KEY command or SETKEY() function with the same key code. For example, assume the F9 key is assigned to look up either customers or vendors depending on the menu option chosen. Listing 17.1 illustrates an example of SET KEY being used to change pop-up programs.

Listing 17.1 Changing SET KEY Table Entries

```
* Program...: Example to change SET KEY table
*
#include "INKEY.CH"
local option:=1
do while !empty(option)
   @ 10,20 prompt "CUSTOMER UPDATE"
   @ 12,20 prompt "VENDOR UPDATE"
   menu to option
   if option == 1
      setkey(K_F9,{ || Cust_Look() } )
      Upd_Cust()
   elseif option == 2
      setkey(K_F9,{ || Vend_Look() } )
      Upd_Vend()
   endif
enddo
```

Removing an entry from the SET KEY table

To remove an entry from the SET KEY table you should use the SET KEY command without specifying a parameter. The syntax is:

```
SET KEY <nKeycode> [TO]
```

This will remove any procedure currently defined for the **<nKeycode>** key. The <nKeycode> may be any number from the ASCII chart or from the INKEY table (Appendix A).

Making INKEY() into a wait state

As we mentioned earlier, the INKEY() function is not a wait state that will access the SET KEY table. We can, however, write a simple UDF to be used in the place of INKEY() which will call INKEY() and test the SET KEY table. The syntax for the INKEY() function which we will replace with Ginkey() is:

```
<nKeycode>  := inkey()
```

<nKeycode> is the ASCII or INKEY code (from Appendix A) that corresponds to the key that was pressed. Our Ginkey() function appears in Chapter 8. See Listing 8.8 there for the function source code.

Writing a pop-up program

Now that we have discussed how to place entries into the SET KEY table, we need to discuss the programming considerations for the procedure that the table will invoke. It is important to remember that the procedure can be invoked anywhere a keystroke is accepted. This means that the pop-up must save the global environment and any database it might manipulate, do its work, and restore the environment and databases exactly to the way they were.

The environment consists of various global settings such as cursor, current color, etc. The global settings are defined via various SET commands. In addition, the environment also includes the current screen. The database environment consists of work areas and their status information. Only databases which the pop-up expects to manipulate need to be saved by the pop-up program. Many pop-up applications do not manipulate database files at all.

Handling the environment

With Clipper you have complete and total knowledge of the state of every SET command via the SET() function. Actually, SET commands have been eliminated completely and are replaced during the preprocessing step with calls to SET(). Here's an example:

```
SET ALTERNATE TO MYFILE.TXT          // Your source code
set(_SET_ALTFILE, "MYFILE.TXT")      // Preprocessor output
```

_SET_ALTFILE is a manifest constant defined in STD.CH along with similar constants for all the other SET commands.

The syntax for the SET() function is:

```
<current setting>  := SET( <nSet_number>,<new value> )
```

<nSet_number> is the numeric code which corresponds to the desired setting. Nantucket stresses in the documentation that this code should never be used directly, but always through the manifest constants defined in STD.CH. The actual numbers may change over the life of the compiler. Table 17.3 contains the set manifest constants.

Table 17.3 SET manifest constants

Constant	*Value type*	*Associated Command/function*
_SET_ALTERNATE	Logical	SET ALTERNATE
_SET_ALTFILE	Character	SET ALTERNATE TO
_SET_BELL	Logical	SET BELL
_SET_CANCEL	Logical	SETCANCEL()
_SET_COLOR	Character	SETCOLOR()
_SET_CONFIRM	Logical	SET CONFIRM
_SET_CONSOLE	Logical	ET CONSOLE
_SET_CURSOR	Numeric	SETCURSOR()
_SET_DATEFORMAT	Character	SET DATE
_SET_DEBUG	Logical	ALTD()
_SET_DECIMALS	Numeric	SET DECIMALS
_SET_DEFAULT	Character	SET DEFAULT
_SET_DELETED	Logical	SET DELETED
_SET_DELIMCHARS	Character	SET DELIMITERS TO
_SET_DELIMITERS	Logical	SET DELIMITERS
_SET_DEVICE	Character	SET DEVICE
_SET_EPOCH	Numeric	SET EPOCH
_SET_ESCAPE	Logical	SET ESCAPE
_SET_EXACT	Logical	SET EXACT
_SET_EXCLUSIVE	Logical	SET EXCLUSIVE
_SET_EXIT	Logical	READEXIT()
_SET_FIXED	Logical	SET FIXED

_SET_INSERT	Logical	READINSERT()
_SET_INTENSITY	Logical	SET INTENSITY
_SET_MARGIN	Numeric	SET MARGIN
_SET_MCENTER	Logical	SET MESSAGE
_SET_MESSAGE	Numeric	SET MESSAGE
_SET_PATH	Character	SET PATH
_SET_PRINTER	Logical	SET PRINTER
_SET_PRINTFILE	Character	SET PRINTER TO
_SET_SCOREBOARD	Logical	SET SCOREBOARD
_SET_SCROLLBREAK	Logical	<varies>*
_SET_SOFTSEEK	Logical	SET SOFTSEEK
_SET_UNIQUE	Logical	SET UNIQUE
_SET_WRAP	Logical	SET WRAP

*_SET_SCROLLBREAK is a global setting which indicates how the Ctrl-S key is handled by the various keyboard commands. If set to .T., Ctrl-S is interpreted as a request to pause the program. If set to .F., Ctrl-S is treated as a cursor movement key, equivalent to the Left Arrow key.

The SET() function returns the current value of the setting.

In addition to the manifest constants in Table 17.3, Clipper also provides a constant called _SET_COUNT, which contains the number of settings currently available in the Clipper syntax.

Save the settings

SaveSets() and RestSets in Listings 12.3 and 12.4 (Chapter 12) can be used to save all global settings.

Saving the screen

Whatever portion of the screen your pop-up is going to operate in must be saved prior to any screen display. Once your pop-up is finished, the screen should be restored, leaving the user right where they were when they pressed the pop-up key.

SAVESCREEN()/RESTSCREEN()

Clipper provides functions to save the screen and restore it. The function to save the screen is SAVESCREEN(). Its syntax is:

```
<cSaved_screen> := SAVESCREEN(<nTopRow>,<TopColumn>, ;
        <nBottomRow>,<nBottomColumn>)
```

<nTopRow>,<nTopColumn> is the upper left corner of the screen to be restored. **<nBottomRow>,<nBottomColumn>** is the lower right corner. The function will return a character string which contains the screen characters and colors from the specified coordinates.

To restore a screen, Clipper provides the RESTSCREEN() function. Its syntax is:

```
RESTSCREEN(<nTopRow>,<nTopColumn>, ;
        <nBottomRow>,<nBottomColumn>, ;
         <cSaved_screen>)
```

<nTopRow>,<nTopColumn> is the upper left corner of the screen to be restored. The **<nBottomRow>,<nBottomColumn>** are the lower right corner. The **<cSaved_screen>** is a character string returned from a SAVESCREEN() function call.

Saving work areas

If your pop-up program is going to work with any .DBF files, you should save the current .DBF file in a stack as well. After all, if your user is updating a customer record and hits a pop-up in the middle, you need to ensure that when the changes are made to the customer file, they are made to the proper record number.

Pushdbf()/Popdbf()

The Pushdbf() function takes care of saving the database status. Its syntax is:

```
Pushdbf( <cAlias> )
```

<cAlias> is an optional alias name, which would default to the current work area. It saves the workarea number, the index order, the record number, and the filter condition.

Popdbf() will restore the database, reposition the record number and index order, and restore the filter command. Its syntax is:

```
Popdbf( <nCount> )
```

<nCount> is an integer indicating how many .DBFs should be restored from the stack. Since you may frequently need to save several databases, necessitating several calls to Pushdbf(), rather than making several calls to the function you can use this count. If you don't pass a number, it will pop only the most recently saved database.

The Pushdbf() and Popdbf() user-defined functions are in Listing 17.3.

Listing 17.3 Pushdbf()/Popdbf()

```
* Program ...: Pushdbf() / Popdbf()
*
static dbfstack_ :={}

function Pushdbf( cAlias )
if cAlias <> nil
   select select(cAlias)
endif
aadd(dbfstack_, Dbfarray() )
return nil
*
function Popdbf( nCount )
local i,nWork:=0,nOrder:=0,nRec:=0,cFilter:=""
local nSize := len(dbfstack_)
nCount := if(nCount=nil,1,nCount)
for i = 1 to nCount
   nWork    := dbfstack_[nSize,1]
   nOrder   := dbfstack_[nSize,2]
   nRec     := dbfstack_[nSize,3]
   cFilter  := dbfstack_[nSize,4]
   if !empty(nWork)
      select (nWork)
      set order to (nOrder)
      goto nRec
      if !empty(cFilter)
         dbsetfilter(cFilter)
      endif
   endif
   Asize(dbfstack_,nSize-1)
   nSize --
next
return nil
*
function DbfArray
local s_:={}
Aadd( s_,select() )       // Save work area number
Aadd( s_,indexord() )     // Controlling index number
Aadd( s_,recno() )        // Current record number
Aadd( s_,dbfilter() )     // Filter condition
return s_
```

Sample pop-up coding shell

Listing 17.4 illustrates a sample shell for a pop-up program. It makes the appropriate calls to various routines to save and restore the environment. It should be used as a basis for your pop-up applications.

Listing 17.4 Sample Pop-up Shell

```
* Program...: Pop-up shell
*
local back_scr := savescreen(0, 0, maxrow(), maxcol())
SaveSets()
Pushdbf()

// Pop-up program's work is done here

// End of pop-up program

Popdbf()
Rest Sets()
restscreen(0, 0, maxrow(), maxcol(),back_scr)
return nil
```

Doing the work

Once you've saved the environment and work areas, your program is free to perform its task. This can be anything from a simple calculation to writing to a log file the notes a user found during testing. Our example applications at the end of the chapter include a pop-up calculator and a calendar.

Saving/restoring GETS

If your pop-up program is going to perform any GETS or READS, it is necessary to save the current GETS. In Clipper 5, this is an extremely simple thing to do. Clipper stores the GETS in a public array called GETLIST. (The array scope is **public** unless your program declares it otherwise). When a new procedure is called, you can create an array for GET just for that procedure. The syntax to create a new GETLIST is:

```
local getlist :={}
```

By declaring GETLIST to be **local** at the beginning of your pop-up procedure, you are telling Clipper to create a new array to hold the GETS in. Any previous GETLIST array will not be overwritten, hence the original GETLIST array is intact and will be restored when your procedure finishes.

Send data back to the calling procedure

Sometimes the pop-up program will need to communicate back to the calling program. This can be accomplished through Clipper's KEYBOARD command. The syntax for the KEYBOARD command is:

```
KEYBOARD<cString>
```

<cString> contains the text of the characters to be returned. The string should also contain any necessary control codes, such as K_ENTER, to allow the GET to continue. The two most common control codes you would need are listed in Table 17.4.

Table 17.4 Common control codes

K_CTRL_Y	Clear the current get field
K_ENTER	Press enter or return key

Listing 17.5 illustrates an example of a selection from a list-box being returned to the keyboard buffer.

Listing 17.5 Keyboard example

```
* Program...: Keyboard example
*
#include "INKEY.CH"
local mcode :=space(8)
memvar getlist
set key K_F9 to Look_Cust

@ 10,10 get mcode pict "!!!!!!!!"
read
if lastkey() <> K_ESC
   *
   * Further processing ......
   *
```

```
endif
return nil

function Look_Cust(cProc,nLine,cVar)
local back_scr := savescreen( 8,20,20,60 )
local send_back
SaveSets()
Pushdbf()
select CUSTOMER
dispbox(8, 20, 20, 60, 2)
Browse(9,21,19,59)
send_back := ( lastkey() <> K_ESC )
Popdbf()
RestSets()
restscreen(8,20,20,60,back_scr)
if send_back
   keyboard chr(K_CTRL_Y)+ ;
          CUSTOMER->id_code+chr(K_ENTER)
endif
return nil
```

Data driven pop-ups

Of course, properly designing and coding a pop-up program is only part of the battle. Once the pop-ups are debugged and in your program, the requests for changes will start: "Can you move the pop-up calculator over to the left?" or "Is it possible to have a red calendar?"

Pop-ups can be coded to get their colors and positions from a .DBF or text file. Using this approach, which is called data-driven programming, you can move pop-ups all over the screen and change their colors without recompiling your program.

Pop-up database

The structure of the pop-up DBF file is shown in Figure 17.2.

Figure 17.2 POPUPS.DBF structure

```
File: POPUPS.DBF
```

Field name	*Type*	*Size*	*Description*
`Pop_name`	`Char`	`12`	`Upper case name of pop-up`
`Pop_color`	`Char`	`25`	`Color string setting`
`Top_row`	`Numeric`	`2`	`Top row`
`Top_col`	`Numeric`	`2`	`Left column`
`Bottom_row`	`Numeric`	`2`	`Bottom row`
`Bottom_col`	`Numeric`	`2`	`Right column`

This file needs not be indexed since even if all 32 set keys were assigned, the entire file would be less than 2,048 bytes. Clipper could read the entire file into memory and allow the LOCATE command to work very quickly.

Dispatch() routine

The Dispatch() user-defined function in Listing 17.7 searches the pop-up database to see if the requested pop-up can be found. If the pop-up is found, the function sets the color and returns a four-element array of screen coordinates. Its syntax is:

```
<array> := Dispatch( <cPop_up_name> )
```

The **<array>** is a four element array containing the screen coordinates corners for the requested pop-up. Table 17.5 lists the constants and array structure.

Table 17.5 Coordinates array structure

Constant	*Element*	*Contents*
TR	1	Upper top row
TC	2	Upper left corner
BR	3	Lower bottom row
BC	4	Lower right corner

Your pop-up function can then save the screen using the coordinates from the array and process its work. Listing 17.6 is an updated version of the pop-up shell from Listing 17.4.

Listing 17.6 Data driven pop-up shell

```
 * Program...: Data driven Pop-up shell
*
#include "INKEY.CH"
#define TR coords[1]
#define TC coords[2]
#define BR coords[3]
#define BC coords[4]
LOCAL back_scr,coords
SaveSets()
Pushdbf()
*
coords := Dispatch( "POPCALC" )
back_scr := savescreen(TR,TC,BR,BC)

// Pop-up program's work is done here

Popdbf()
RestSets()
restscreen(TR,TC,BR,BC,back_scr)
return nil
```

The code for the Dispatch function is in Listing 17.7.

Listing 17.7 Dispatch() Function

```
* Program...: Dispatch function
*
function Dispatch( cPop_up )
local sarr_ :={}
use POPUPS new
locate all for POPUPS->pop_name == upper(cPop_up)
if found()
  setcolor(POPUPS->pop_color)
  aadd(sarr_,POPUPS->top_row)
  aadd(sarr_,POPUPS->top_col)
  aadd(sarr_,POPUPS->bottom_row)
  aadd(sarr_,POPUPS->bottom_col)
endif
use
return sarr_
```

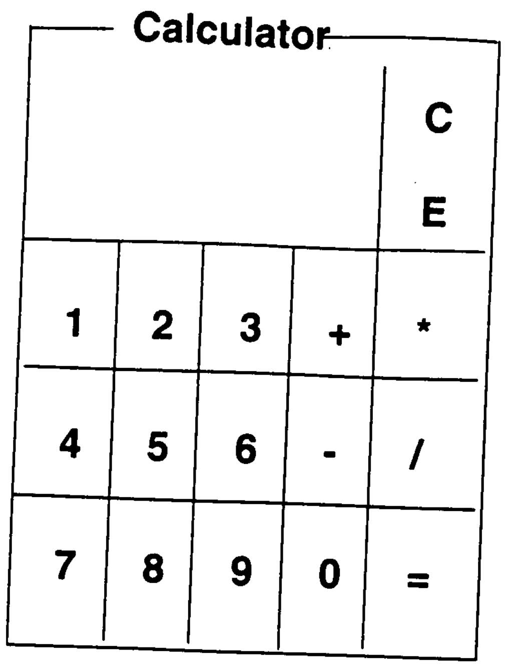
Pop-up examples

In this section we will provide some example pop-up programs that you might frequently need in your applications. The program can be used as written or adapted to provide additional pop-up utilities.

Pop-up calculator

Listing 17.8 illustrates a simple pop-up calculator. This program displays a calculator keypad and lets the user enter mathematical operations. The results are calculated and can be pasted back to the calling GET if desired. Figure 17.3 shows the calculator and the operations it supports.

Figure 17.3 Calculator Screen and Operations



C	Clear the number
+	Add number to total
-	Subtract from total
*	Multiply two numbers
/	Divide two numbers
=	Paste result to GET

Listing 17.8 Pop-up calculator

```
 * Program...: Popcalc
 *
 #include "INKEY.CH"
 #include "BOX.CH"
 function Popcalc(cProc,nLine,cVar)
 local back_scr := savescreen(2,5,11,25)
 local adding:=.t.,first_time:=.t.
 local calc_amt:=0,running:=0,calc_op:=" "
 local getlist:={}                              // Protect get stack
   SaveSets()
/**********************************
*    Paint calculator on screen    *
**********************************/
@ 02,05,11,25 box B_SINGLE_DOUBLE+" "
@ 02,06 say "  Calculator "
@ 03,06 say "                  | C "
@ 04,06 say "                  | E "
@ 05,06 say " ────────────────── "
@ 06,06 say " 1 | 2 | 3 | + | * "
@ 07,06 say " ────────────────── "
@ 08,06 say " 4 | 5 | 6 | - | / "
@ 09,06 say " ─────────────────── "
@ 10,06 say " 7 | 8 | 9 | 0 | = " /
****************************
*   Main calculator loop   *
****************************/
do while adding
  @ 3,9 say running picture "99999999.99"
  if !first_time
     @ 4,7 get calc_op  picture "!" valid calc_op$"+-/*=C X"
  endif
   @ 4,9 get calc_amt picture "99999999.99"
   read
   first_time := .f.
   ******************************
   *     Process key stroke     *
   ******************************
   if lastkey() <> K_ESC
      *
      do case
      case calc_op == "C"                    && Clear key
         store 0.0 to running,calc_amt
      case calc_op == " "
         running := calc_amt
      case calc_op == "+"                    && Add
         running += calc_amt
      case calc_op == "*"                    && Multiply
         running *= calc_amt
      case calc_op == "-"                    && Subtract
         running -= calc_amt
      case calc_op == "/"                    && Divide
         if calc_amt > 0
            running /=  calc_amt
         endif
      case calc_op == "X"       //  X =   Paste the result
         if valtype(cVar) = "N"
           keyboard chr(K_CTRL_Y)+str(running)+chr(K_ENTER)
         endif
```

```
            adding := .f.
        endcase
        *
        calc_amt := 0.0
        *
        else
            adding = .f.
        endif
enddo
RestSets()
restscreen(2,5,11,25,back_scr)
return NIL
```

Pop-up calendar

Listing 17.9 illustrates a simple pop-up calendar. This program displays a calendar for the current month. If the user presses the page up key, the next month's calendar will be displayed. Pressing page down will display the previous month's calendar. The home and end keys will respectively display the first and last month of the year.

Listing 17.9 Pop-up Calendar

```
* Program...: Popcalend
*
#include "INKEY.CH"
#include "BOX.CH"

function Popcalend(cProc,nLine,cVar)
local back_scr := savescreen(8,20,17,50)
local wmonth   := month(date()),nkey:=1 /
local oldcurs := setcursor (0)
********************************
*    Paint calendar on screen    *
********************************/
@ 08,20,17,50 box B_SINGLE_DOUBLE+" "
draw_cal( wmonth )
```

```
/****************************
*    Main calendar loop    *
****************************/
do while !empty(nKey)
   nKey := inkey()
   do case
   case nKey == K_ESC
      nKey := 0
      loop
   case nKey == K_PGUP
      if ++wmonth > 12
         wmonth :=1
      endif
      draw_cal( wmonth )
   case nKey == K_PGDN
      if --wmonth < 1
         wmonth :=12
      endif
      draw_cal( wmonth )
   case nKey == K_HOME
      wmonth :=1
      draw_cal( 1 )
   case nKey == K_END
      wmonth := 12
      draw_cal( 12 )
   endcase
enddo
restscreen(8,20,17,50,back_scr)
setcursor(oldcurs)
return nil
*
function draw_cal(nMonth)
local jj,tt,temp:=str(nMonth,2)+"/01/91"
local start := dow(ctod(temp))-1,pday:=0
local templ := str(if(nMonth<12,nMonth+1,1,1)2) + "01/91"
local last := day(ctod(templ)-1)
```

```
@  9,21 clear to 16,49
@  9,21 say padc(cmonth(ctod(temp)),28)
@ 10,21 say " Sun Mon Tue Wed Thu Fri Sat"
for jj=1 to 6
   Devpos(10+jj,21)
   for tt=1 to 7
      if ((tt<start+1) .and. jj=1) .or. pday >= last
         ?? space(4)
      else
         ?? str(++pday,4)
      endif
   next
next
return nil
```

Summary

After reading this chapter you should understand how to place and remove entries from the Clipper keystroke table. You should also be aware of the considerations of coding pop-up programs. Finally, you should be comfortable with the pop-up examples in case your users need a calculator or calendar or you wish to create your own pop-up utility.

CHAPTER 18

Working with .DBF Files

Clipper is designed to manipulate files stored in dBASE file format as originally defined by Ashton-Tate's dBASE product. This format provides a simple file structure that allows the definition of fields within a record. These fields may be character (fixed length), date, numeric, logical (y/n), or memo (variable length character data).

This chapter explores the structure of the .DBF file. It also describes the tools that Clipper provides to work with .DBF files. Finally, we will discuss data compression techniques that can be used to reduce the amount of disk storage space that a .DBF file will use.

Clipper's device drivers

While it's not the case currently in Clipper 5, it is the future goal of Nantucket to provide a strategy for Clipper to work with a variety of database formats. These formats might include .DBF, Paradox, R:BASE, and other files. The concept is known as Replaceable Device Drivers, or RDDs for short.

Clipper will allow each work area to be controlled by a different device driver, so one work area might have a Paradox DB file, and another area a .DBF file. The same Clipper command syntax could be used transparently with whatever type of file is in the work area.

Currently, the only device driver provided by Nantucket is the DBFNTX driver, which works with .DBF files and .NTX index files. However, the commands and functions discussed in this chapter will be used exactly the same regardless of the device driver controlling the work area.

.DBF files

.DBF is an acronym for Database File. It is a simple file structure which consists of a header portion and a data portion. Within the header portion are a number of field definitions, which provide the layout of each individual record. When a record is selected, the data bytes of that record are mapped into the fields and made available to the application program.

File structure

Table 18.1 shows the internal structure of a .DBF file. Fields that are one byte in size are converted to numbers using the ASC() function. Two-byte fields are converted using the BIN2W() and four-byte fields use the BIN2L() function.

Table 18.1 DBF structure

Start	*Bytes*	*Contents*
1	1	Signature byte: 03=no memo field,131=memo field
2	1	Year of last update
3	1	Month of last update
4	1	Day of last update
5	4	Number of records
9	2	Data offset
11	2	Record size
13	20	Filler, unused space

The field information array starts here. Each entry is 32 bytes long with the structure shown below:

Start	Bytes	Contents
1	11	Field and CHR(0) terminator
12	1	Type, (C)har,(D)ate,(L)ogical,(M)emo,(N)umeric
13	4	Not used
17	1	Field size
18	1	Field decimals
19	14	Not used

The final field definition will be followed by a CHR(13). Notice that the header portion of the structure does not keep a count of number of fields.

Checking for memo files

If a .DBF file contains a memo field then a second file must exist which has the same name, but an extension of .DBT. This is where the text to the memos are stored. When Clipper opens a .DBF file, it checks to see that the corresponding memo file exists. If the memo file does not exist, the following run-time error occurs:

```
Open Error: <filename.DBT> DOS Error 2
```

DOS Error 2 is Clipper's graceful way of saying that the file cannot be found. The function in Listing 18.1 uses the low-level files functions (see Chapter 24) to check if a .DBT is needed. If a .DBT file is needed and one exists, the function returns true. If one is needed, but does not exist, false is returned. This function allows your program to keep control and decide upon a course of action for missing memo files. Its syntax is:

```
<logical>  := Safe2open( <file_name> )
```

The .DBF extension is not passed as part of the file name.

Listing 18.1 Safe2open

```
function safe2open(file_name)
local handle, retval:=.t., first_byte, buffer:=" "
local fn := trim(file_name)
if (handle := Fopen(fn+".DBF")) > -
   fread(handle,@buffer,1)                   // Read first byte
   if (first_byte := asc(buffer)) == 131 // Memo file ?
      retval := file( fn+".DBT")           //  Does the file exist?
   endif
   fclose(handle)                         // Close the file
endif
return retval
```

Protecting files from dBASE use

Since the .DBF file structure was designed to be used in an interpretive, interactive mode, a user can easily access a .DBF file by using dBASE or similar products. The user can easily delete records or change key values. If the user adds records or changes key values, then the Clipper index files will not be properly updated.

To keep a casual user from accessing the .DBF files under dBASE, we can use Clipper's low-level file functions to prevent dBASE interpreters from recognizing the file. The first byte of a .DBF file is a signature byte identifying it as a dBASE file. If this byte is changed, dBASE will not recognize the file as a valid file. The Protect() function in Listing 18.2 changes the first byte of the file to a different byte, which will prevent dBASE from using the file. The function syntax is:

```
Protect( <file_name> )
```

The **file_name** should not include the .DBF extension. However, this function is not foolproof. A user familiar with the DOS debug program or similar programs could easily change the first byte back to an 03 or a 131.

Listing 18.2 Protect()

```
#include "FILEIO.CH"

function Protect(file_name)
local handle, first_byte, buffer:=" "
local fn:=trim(file_name)
if (handle := Fopen(fn+".DBF",FO_READWRITE)) > -1
   fread(handle,@buffer,1)          // Read first byte
   first_byte := asc(buffer)
   fseek(handle,0,0)                // Go to top of file
   if first_byte == 131             // If memo file field
      fwrite(handle,chr(27),1)      // write 27 as the first
   else                             // byte, otherwise write
      fwrite(handle,chr(26),1)      // a 26.
   endif
   fclose(handle)                   // Close the file
endif
return nil
```

Listing 18.3 provides a function that does the reverse of the Protect() function. When you unprotect the file, it is available for use by Clipper and dBASE.

Listing 18.3 Unprotect()

```
#include "FILEIO.CH"

function Unprotect(file_name)
local handle,first_byte,buffer:=" "
local fn:=trim(file_name)
if (handle := Fopen(fn+".DBF",FO_READWRITE)) > -1
   fread(handle,@buffer,1)    // Read first byte
   first_byte := asc(buffer)
   fseek(handle,0,0)          // Go to top of file
   if first_byte == 27        // Is a memo file needed?
      fwrite(handle,chr(131),1)
   else
```

```
      fwrite(handle,chr(03),1)
   endif
   fclose(handle)                   // Close the file
endif
return nil
```

If keeping the users away from the files is a necessity, you should use Unprotect() at the beginning of your Clipper program and Protect() at the end. You should also include code in your error handling program to protect databases in the event of a system error. This way, if the program aborts due to an error, the databases will still be protected from dBASE users.

If your application is running on a network, another user could alter the file with dBASE while the Clipper application is running. Unfortunately dBASE might not recognize Clipper's file or record locks.

Work areas

To manage all the information associated with the .DBF file, Clipper creates a table in memory where information about open files is stored. Each entry in this table is called a work area. A .DBF file must be assigned to a work area in order for Clipper to work with that file. Clipper allows up to 250 work areas. When a program first starts, all work areas in the table are empty and the current work area is set to one.

The work area table maintains the following information for each active file:

Alias()	Work area name
Bof()	Is file pointer at beginning of file?
Dbfilter()	The filter expression active for this file
Dbstruct()	An array of field names/types/sizes.
Dbrelation()	The linking expression to related files
Dbrselect()	Work area number to which this work area is related
Deleted()	Is current record deleted?
Eof()	Is file pointer at end of file?

Fcount()	Number of fields in the file
Field()	Field name
Found()	Result of last SEEK or LOCATE command
Header()	Size of header in bytes
Indexkey()	Key expression for an index
Indexord()	Controlling index number
Lastrec()	Number of records in the file
Lupdate()	Last date .DBF file was updated
Recno()	Record size in bytes
Select()	Work area number
Used()	Is an open file in a work area?

To open a .DBF file, you must select the work area you wish to put this file in. The SELECT command and the DBSELECTAREA() function allow you to choose a work area. The syntax is:

```
//  SELECT command

SELECT <nWorkarea>

// DBSELECTAREA() function

DBSELECTAREA( <nWorkarea> )
```

If the work area is between 1 and 250, that work area becomes the current work area. Clipper does not check to see if that work area is empty or not. If you select 0 as the work area, Clipper selects the next available empty work area.

The SELECT command and DBSELECTAREA() function also allow you to specify an alias name rather than a work area number. The syntax is:

```
// SELECT command using alias name
SELECT <cAlias>

//  DBSELECTAREA() function

DBSELECTAREA( <cAlias> )
```

Where possible, you should try to use the second form of the select command. produces more readable code. For example,

```
SELECT CUSTOMER
```

instead of:

```
SELECT 5
```

Opening a file

Once you've selected a work area, you may use the USE command or the DBUSEAREA() function to open a .DBF file in that work area. The syntax is:

```
// USE command

USE <cDbf_file> [ INDEX <cNtx_file(s)> ]
  [ALIAS <cAlias>] [SHARED|EXCLUSIVE] [NEW] [READONLY]
  [VIA <cDriver>]

// Dbusearea() function

DBUSEAREA( [<lNew>],[<cDriver>],<cDbf_file>, ;
           [<cAlias>],[<lShared>],[<lReadonly>] )
```

<cDbf_file> is the name of the database file you wish to open. If another file is already in use in the work area, that file will be closed before the new file is opened. The **cDbf_file** may be a literal value or a character string enclosed in parentheses.

<cNtx_file(s)> is a list of index file names to open in the work area. This list may contain up to 15 index files.

Although this syntax is available, in a network environment the preferred syntax is to use the SET INDEX TO command. When you're on a network, you need to check if the file was successfully opened before assigning indexes to the work area. The SET INDEX command allows you to separate the file opening and the index opening into two commands. This allows you to check the file's open status first.

Notice that the DBUSEAREA() function does not provide for opening of indexes. This would have to be done using the Dbsetindex() function we'll describe later.

The ***<cAlias>*** keyword allows you to specify a different alias name for the file. If you do not specify an alias, the file name will be used as the work area alias. An alias can be up to eight characters long. It is also **not** a good practice to use a single letter as an alias name. Using a single letter introduces the possibility of your alias name conflicting with Clipper's built in alias names.

The ***<new>*** keyword or setting ***<lNew>*** to **.t.** informs Clipper to open this file in the next available work area. If new is not specified, the file is opened in the current work area. **New** has the same effect as specifying SELECT 0 before the USE command.

The ***<shared>*** or ***<exclusive>*** keyword determines if other users will be able to access this file on a network. The SET EXCLUSIVE command allows you to specify the default, which is normally exclusive. However, Nantucket considers SET EXCLUSIVE to be a compatibility command, which means it might not be there in future releases. It is best to explicitly state the shared or exclusive mode when you open the file.

The ***<lShared>*** flag on the DBUSEAREA() function can be set to **.t.** for shared access, or **.f.** for exclusive file use.

If the keyword ***<Readonly>*** is specified, the file will be opened for reading only. This allows password and other key tables to be opened, but not updated. Setting ***<lReadonly>*** to **.t.** has the same effect as specifying the **readonly** keyword. The default is to open the file for reading and writing access.

The keyword ***via <cDriver>*** or the ***<cDriver>*** option is used to specify a replaceable database driver when the file is opened. Replaceable device drivers are an exciting future enhancement planned for Clipper. By using a different driver, your Clipper application will be able to open a .DBF file in one work area, and a Paradox file, for example, in another. The Clipper commands and functions for manipulating work areas will work on either file transparently. The database driver will translate function calls into the appropriate action for the type of file.

Currently, the only driver is "DBFNTX", although others are expected in the future. If the driver specified using the **via** option is not linked into the application, an unrecoverable run-time error will occur. The driver name is a character string and may be a variable or a constant value. If a constant value is used, it must be enclosed in quotes.

DBSETDRIVER()

The DBSETDRIVER() function returns the name of the default database driver. This driver is used whenever a database file is opened without using the **via** option. The function can also be used to set the default driver to a different driver. Its syntax is:

```
cCurrent_driver := DBSETDRIVER( [<cDriver>] )
```

The ***<cCurrent_driver>*** is the name of the default driver. If **<cDriver>** is supplied, the default driver is changed to the new driver. If the selected driver is not available to the application, the default driver is not changed. As mentioned above, the current driver provided by Nantucket is DBFNTX.

ALIAS()

The ALIAS() function returns the database name for any work area. If an **alias** was specified in the USE command, the alias will be returned rather than the database name. The syntax is:

```
<cName> := ALIAS( <nWork_area> )
```

The **nWork_area** is a number in the range of 1 through 250. If not supplied, the current work area will be used. ALIAS() can be used to present a list of files for the user to select from. The Pickalias() function shown below returns an array of open alias names.

```
function Pickalias
local alist:={},jj
for jj:=1 to 250
   if !empty(alias(jj))
      aadd(alist,alias(jj))
   endif
next
return alist
```

USED()

The USED() function returns a logical value indicating whether or not a file is opened in a particular work area. Its syntax is:

```
<lUsed> := USED()
```

The function only checks the current work area. To check a different work area number, you would need to first issue a SELECT command. For example:

```
select 20
? USED()
```

If you know the work area's alias, you can use the extended expression syntax as shown in the example:

```
use CUSTOMER new
select 20
? CUSTOMER->( used() )
```

SELECT()

The SELECT() function returns the work area number for any alias. It is the inverse function to ALIAS(). Its syntax is:

```
<nWork_area> := SELECT( <cAlias> )
```

<nWork_area> will be a number in the range of 1 through 250. If an alias is not supplied, the current work area's alias will be used. If the alias specified is not found, the function will return zero.

The SELECT() function can be used to save the current work area and later restore it. The Pushalias() and Popalias() functions shown in Listing 18.4 illustrate this functionality.

Listing 18.4 Pushalias() and Popalias()

```
// Note: Compile with /n
static astack:={}                          // file-wide static array

function Pushalias
aadd(astack,select())
return nil
```

```
function Popalias
local x := astack[ len(astack) ]
asize(astack,len(astack)-1)
select (x)
return nil
```

This program requires the compiler switch **/n** since a **static** array is being used.

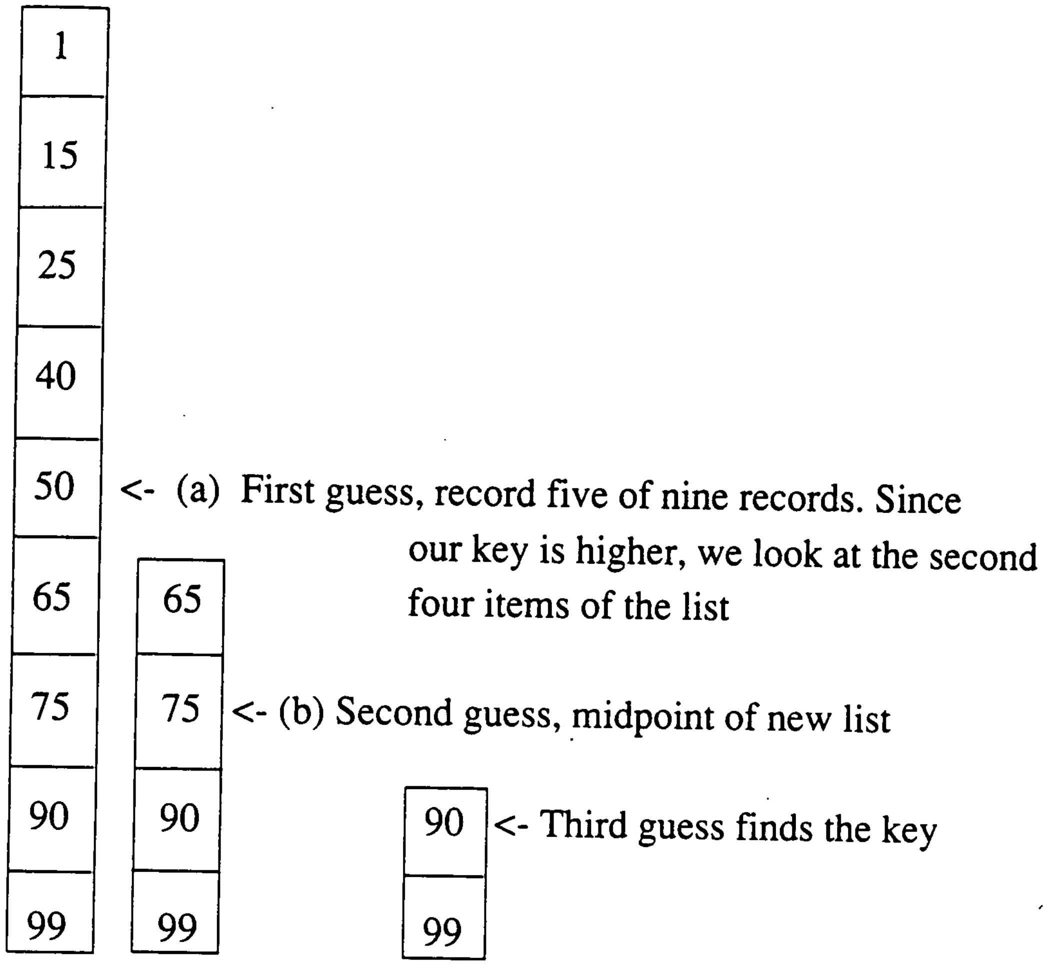
The two functions can be useful in designing pop-up programs. A pop-up program might change the work area. By starting the function with a Pushalias() call and ending with a Popalias() call, your pop-up program is free to move around in different work areas. The use of a stack allows pop-up programs to be called from within other pop-up programs. See Chapter 17 for more information on pop-up programming.

Using indexes

An index is a supplemental file to a .DBF file. It is used to speed up access and to provide a logical ordering to the file. The index file consists of a list of key expressions and record pointers. The expressions are stored in a b-tree structure which allows rapid lookup and transversal.

A b-tree structure is an expansion of the binary search technique. The binary search is a quick method for searching an ordered list of values. It begins by examining the value at the midpoint of the list. If that value is the key we are searching for, the search is done. If that value is greater than the key we are searching for, we find a new midpoint in the lower half of the list. If the value is lower than the key, we compute a new midpoint in the upper half of the list. The process is repeated until the key is found. Figure 18.1 illustrates a binary search being used to find the value of 90 in a list of 100 records.

Figure 18.1 Binary search, seeking value 90


1
15
25
40
50
65
75
90
99
<- (a) First guess, record five of nine records. Since our key is higher, we look at the second four items of the list
65
75
90
99
<- (b) Second guess, midpoint of new list
90
99
<- Third guess finds the key

The b-tree expands the technique by using pages. A page consists of as many entries as possible. To determine the number of entries per page, Clipper takes the page size of 1024 bytes and divides it by the size of the index key plus an eight-byte pointer.

Each entry in a page consists of the key value, a record number, and pointers. When a page is read into memory, all the entries are compared with the key. If an entry is found, the database is moved to that record number. If an entry is not found, Clipper uses the pointers to determine the next index page to read from the disk. The process is repeated until a key is found, or the pointer values indicate that there are no more pages.

Clipper index files cram as many pointers as possible into each page. This helps to reduce the number of disk reads needed to find the record. The fewer disk reads, the faster the lookup takes place. When designing your index keys, keep in mind that smaller keys allow more keys per page, and hence fewer disk reads. This results in faster index lookups.

A database file can have up to fifteen indexes. All indexes must be specified when the file is opened to ensure that they will be properly updated during database operations. One index is considered the controlling index. This index determines the order of the records and is the index against which seeks and finds are checked. If the controlling index is zero, the database is ordered by record numbers.

Clipper supports two types of indexes. The first index is Clipper's own indexing scheme, which has a file extension of .NTX. This is the default indexing scheme to use. The structure for a Clipper index file is listed in Table 18.2.

Table 18.2 NTX file structure

	Header	1024 Bytes long
Start	Size	Contents
1	2	Signature byte 03 =Clipper index file
3	2	Clipper indexing version number
5	4	Offset in the file of first index page
9	4	Offset to list of unused pages
13	2	Key size + 8 bytes for pointers
15	2	Key size
17	2	Decimal places in key, if numeric
19	2	Maximum entries per page
21	2	Minimum entries per page
23	256	Key expression followed by a chr(0)
279	1	1 if unique index, 0 if not
280	744	Filler

Clipper also supports dBASE-compatible indexes. This allows your application to work with index files the user updates during an interpretive session in dBASE. These index files have a file extension of .NDX. The structure for a dBASE compatible index is listed in Table 18.3.

Table 18.3 NDX file structure

Header 512 bytes long		
Start	Size	Contents
1	4	Record number of root page
5	4	Number of 512 byte pages in the file
9	4	Filler
13	2	Size of index key
15	2	Maximum number of keys per page
17	2	Key type 1=numeric, 0=character
19	4	Size of key records
23	1	Filler
24	1	1 if unique index, 0 otherwise
25	488	Key expression

If you choose to work with dBASE-compatible indexes, you will need to link the dBASE index driver into your program. This .OBJ file, if included in your link cycle, instructs Clipper to work with the dBASE .NDX files instead of Clipper's .NTX format. The driver is available from Nantucket to all registered Clipper owners.

Creating INDEX files

The INDEX command or the DBCREATEINDEX() function are used to create a supplemental index file. The syntax is:

```
// INDEX command

INDEX ON<expression> TO <cFile_name> [UNIQUE]

// DBCREATEINDEX() function

DBCREATEINDEX( <cFile_name>,<expression>,[bIndex],;
               [<lUnique>] )
```

<expression> can be any valid Clipper expression, including user-defined functions. The expression must evaluate to the same length for each record in the database. For example, the following index command will create an index file that will most likely get corrupted very quickly.

```
INDEX ON trim(Name) TO cust001
```

Unless each name in the database is exactly the same size, indexing on the TRIM() will produce variable-length index keys. Clipper does not support variable-length index keys. This becomes even more important when using user-defined functions as index keys.

When using a user-defined function be sure to keep the function as small as possible. The INDEX ON command will process every record in the database. If your user-defined function is twenty lines of code and there are 500 records in the database, 10,000 lines of Clipper code will be executed by the INDEX ON command.

The optional ***<bIndex>*** parameter is a code block which should be used to create the index. If it's not supplied, Clipper will use the macro expansion of the ***<expression>*** to create the index.

The use of the ***<bIndex>*** code block and the DBCREATEINDEX() function allow indexes to be created without causing Clipper to call the macro processor.

It is important to realize that even if a code block is provided, the ***<expression>*** parameter is still required. This expression is written into the index file header. Since Clipper cannot write a code block to disk, the expression must be provided to the function.

<cFile_name> is the name of the index file to be created. If it already exists on disk, it will be overwritten. The file name will automatically be given the extension of .NTX if no extension is specified.

The ***<unique>*** keyword or setting ***<lUnique>*** to **.t.** instructs Clipper to write only one entry for each occurrence of a key. If a key value returned by the index expression appears more than once, only the first occurrence will be written into the index.

Unique indexes can also be created using the SET UNIQUE command. Its syntax is:

```
SET UNIQUE  <ON|OFF|<lToggle>
```

The normal default for index creation is non-unique indexes. The SET UNIQUE command allows you to set the default value. Nantucket considers SET UNIQUE to be a compatibility command and therefore it is not guaranteed to be in future releases of the compiler.

Indexing Dates. When indexing date fields, it is better to use the DTOS() function rather than the DTOC() function. DTOS() converts the date to a consistent format, YYYYMMDD, which will produce sequential ordering. DTOC() is dependent upon the setting of the SET DATE command, which can vary between indexing and updating the index.

DESCEND(). Indexing normally proceeds in ascending order. You can create a descending index by enclosing the index expression as a parameter to the descend function. DESCEND() returns a complemented form of the expression passed, which will cause the index to appear in descending order. The syntax of DESCEND() is:

```
<expression>  := DESCEND( <expression> )
```

The type of the returned expression will be the same type as the type of the parameter, except if the parameter is a date. In that case, the DESCEND() function will return a numeric value which complements the date passed.

In order to understand how DESCEND() works, let's look at two approaches we could use to write our own descend function. The purpose of DESCEND() is to switch the order of the data from LOW-TO-HIGH to HIGH-TO-LOW. One simple approach would be to multiply the data by -1. This has the impact of making the highest numbers the lowest and vice-versa. The drawback of this approach is that it only works on numbers. For numeric data, this is the approach Clipper's DESCEND() function uses.

The second approach is to take the number and subtract it from a much larger number. The larger the starting number is, the smaller the result when we subtract it. For example,

```
big := 10000

? 1500, big - 1500   // displays   1500   8500
? 3000, big - 3000   //             3000   7000
? 4500, big - 4500   //             4500   5500
```

To apply this approach to characters, remember that the computer views character strings as an array of ASCII numbers. To create a descending order string, Clipper subtracts each character's ASCII code from 255, and returns the CHR() of the result. This approach solves the descend problem, but also produces some strange-looking character strings.

The listing below shows some examples of the DESCEND() function.

```
index on descend(trans_date)  to trans.ntx
seek descend( date() )

? descend("CLIPPER") // displays gibberish
? descend(1500)      // displays -1500
```

Reindexing files

The REINDEX command or the DBREINDEX() function recreates all open indexes within a work area. The syntax is:

```
// REINDEX command

REINDEX

// DBREINDEX() function

DBREINDEX()
```

REINDEX processes the list of open index files for a particular work area and rebuilds each one. It does not rebuild the header, only the index body. If the index file header has been corrupted, reindex will not fix it.

The following example reindexes all open indexes in the customer work area.

```
use CUSTOMER index CUST1,CUST2 new
reindex
```

You can also use the DBREINDEX() function to rebuild indexes in an unselected work area. For example,

```
CUSTOMER->( dbreindex() )
```

Keep in mind that if an index file is corrupt, reindex will not repair it.

Specifying indexes

The SET INDEX TO command or the DBSETINDEX() function are used to specify the list of index files that should be associated with a particular database file when it is used. It is the same as specifying an index list in the USE command. The syntax for SET INDEX is:

```
SET INDEX TO <cIndex_file(s)>
```

The syntax for DBSETINDEX() is:

```
DBSETINDEX( cIndex_file,[<bExpression>] )
```

<cIndex_file> is the name of the disk file containing the index. The SET INDEX command allows multiple files to be specified at one time, while the DBSETINDEX() function allows only one index at a time to be specified. A second call to SET INDEX releases the old list of index files and establishes a new list. Calls to DBSETINDEX() are additive, so several indexes can be opened by calling DBSETINDEX() several times.

The optional ***<bExpression>*** is a code block which, if supplied, will be evaluated to update index keys in the index. If it's not supplied, the index header contains the character expression of the index, and this expression will be macro-expanded to process index key entries. See Chapter 8 for more information on code blocks.

Note that Clipper will not compare ***<bExpression>*** with the character string expression from the index file. It is entirely possible to supply a code block that is not consistent with the index key in the .NTX file header. Doing so, however, will almost guarantee corrupt indexes, since some keys will be different from others. If the keys are different lengths, corrupt indexes are only a matter of time. If the keys are the same length, but have different contents, you could have some very subtle bugs to trace down.

Once the index list is set, all indexes in the list will be updated whenever a database operation is performed. If there are no index files listed, the SET INDEX command closes all open indexes in the current work area.

The list of index files can be literal names or character strings surrounded by parentheses. If the character string evaluates to spaces or a NULL it is ignored.

Listing 18.5 shows some examples of using SET INDEX and DBSETINDEX().

Listing 18.5 Set index examples

```
local mfile:="VENDOR",mfile1:="", mfile2:="", mfile3:=""
local ixarr_:={ "LINE1","LINE2" },k

use CUSTOMER new
set index to CUST1,CUST2,CUST3

mfile1 := "VEND1"
mfile2 := "VEND2"

use (mfile)
set index to (mfile1),(mfile2),(mfile3)

use INVOICES new
dbsetindex( "INV01" )
dbsetindex( "INV02" )

use LINEITEM
for k:= 1 to len(ixarr_)
   dbsetindex( ixarr_[k] )
next
```

To release all indexes from a work area, you can use the SET INDEX TO command followed by no file names. The DBCLEARINDEX() function performs the same function. Its syntax is:

```
DBCLEARINDEX()
```

SET ORDER TO

Once indexes are established for a work area, the SET ORDER TO command or the DBSETORDER() function allow you to switch the controlling index. The ***controlling index*** is the index which is considered active. The database will be displayed in this order and all SEEK and FIND commands will use this index. The syntax is:

```
// SET ORDER TO command

SET ORDER TO <nOrder>

// DBSETORDER() function

DBSETORDER( <nOrder> )
```

The ***<nOrder>*** may be any number from zero to the number of index files for the work area, a maximum of fifteen. If no number is specified, the order will be set to zero. Zero is natural order, which is ordered by record number. It is also the physical order in which the file is written to disk.

No matter which index is the controlling index or if the file is in natural order, all open indexes will be updated during database operations.

INDEXKEY()

The INDEXKEY() function returns the key expression on which the specified index was created. The index must be open either by a USE or a SET INDEX command. The syntax for INDEXKEY() is:

```
<cExpression> := INDEXKEY(<nOrder>)
```

<nOrder> indicates which index file from the list the expression should be returned for. If ***<nOrder>*** is zero, the expression of the controlling index is returned.

INDEXKEY() can be used to present a list of sorted views of the database for a user to choose. Listing 18.6 provides a function which will return an array of currently open index expressions in a work area.

Listing 18.6 Ntxkeylist()

```
function Ntxkeylist
local ixlist:={},jj,kk:=indexord()
for jj:=1 to 15
   if !empty(indexkey(jj))
       aadd(ixlist, indexkey(jj)+if(kk==jj," √ ","   "))
   endif
next
return ixlist
```

The controlling index is flagged with a check mark when the array is returned.

INDEXORD()

The INDEXORD() function returns an integer value which corresponds to one of the indexes specified on the USE or SET INDEX command. The value returned by INDEXORD() is the controlling index. The controlling index is the index file that determines the database order and that SEEK commands will be issued against. Clipper will automatically update all open indexes, but only one index may be in control at a time. The SET ORDER command allows you to switch among multiple indexes.

If there is no controlling index, the INDEXORD() function returns a zero. If there is no controlling index, the records are accessed in natural order, that is, the physical order they are in the file.

The syntax for INDEXORD() is:

```
<nControl> := INDEXORD()
```

The INDEXORD() function can be used to save the current controlling index and restore it after some commands are performed. For example:

```
use CUSTOMER new index cust_id,cust_nam
nCurrent := indexord()          // 1 - cust_id index
set order to 2                  // switch to name order
list id_code,name,address,city,state to print off
set order to (nCurrent)         // restore original order
```

INDEXEXT()

The INDEXEXT() function returns a three character code indicating the index driver linked into the program. The default code, which is also the file extension, is NTX. If the dBASE compatible index driver is linked into your application, NDX will be returned.

The syntax for INDEXEXT() is:

```
<cExtension> := INDEXEXT()
```

The INDEXEXT() function should be used when testing for the presence of index files. For example:

```
if file("CUST1."+indexext())
   set index to cust1
else
   index on id_code to cust1
endif
```

Relating files

In database applications, files may need to share information between them. For example, on a purchase order the vendor's name and address information usually appears. This information would also be stored in the VENDOR file. If the address is stored in both files, there is the risk of data being updated in one file and not the other. In addition, the redundant information is using up space on the hard disk.

A solution that can be implemented in Clipper is to relate the two files. This is done by choosing a common key between them. A common key between a purchase order file and the vendor file would probably be the VEND_ID code. The purchase order file would include a field for VEND_ID with no additional vendor information. When the vendor name and address is needed, the programmer would ask the system to go into the vendor file, find the vendor with the appropriate VEND_ID, and display that vendor's information.

Clipper provides commands and functions to make file relations easier to work with.

SET RELATION and DBSETRELATION()

The SET RELATION command or the DBSETRELATION() function is used to link two or more work areas together by a common key field or by record numbers.Both work areas must be open. The syntax is:

```
// SET RELATION command

SET RELATION TO <expression| recno()> INTO <cAlias>
[ADDITIVE]

//  DBSETRELATION() function

DBSETRELATION( <nArea|cAlias>,<bExpr>,[<expression>] )
```

<expression> is the key field or expression on which the link should be performed. When Clipper encounters this field, it is assumed that the related database has been indexed on this field and that this index is the controlling index. This expression is returned by the DBRELATION() function.

If ***<recno()>*** is used, the files will be linked together by their record numbers. The use of record numbers to link files together should be carefully considered. If the database is packed, record numbers may change, which would corrupt the relation. In addition, if two files are related by record numbers, the second file must not have any active indexes. Either no indexes are opened for that file or a SET ORDER TO 0 command has to be issued in that work area.

<bExpr> on the DBSETRELATION() call is a code block that is evaluated to determine the value used to reposition the related file. By specifying a code block instead of a character expression, macro-expansion is not performed. The ***<expression>*** is still used, however, to return a value from the DBRELATION() function.

<cAlias> or ***<nArea>*** indicates the work area alias to which the current work area should be related. It may be a literal value or an extended expression enclosed by parentheses or it may be a numeric work area reference. It cannot be the current work area.

The **ADDITIVE** keyword allows additional relations to be created in the same file. If ADDITIVE is not specified, a SET RELATION command will erase previous relations before creating the new link. The DBSETRELATION() function is additive, that is, it does not clear any existing relations.

SET RELATION TO with no parameters removes the link between files. You may also use the function DBCLEARRELATION() to remove the links between files. Its syntax is:

```
DBCLEARRELATION()
```

More than one relation can be specified on a single SET RELATION command. For example,

```
set relation to VEND_ID into vendor, ZIP_CODE into zips
```

The code below illustrates a relation to link the purchase order file to the vendor database.

```
use PURCHORD new
use VENDOR new index VEND_ID
select PURCHORD
set relation to VEND_ID into VENDOR
```

Relations are numbered sequentially as they are applied to a file.

This occurs whether the relations are established on the same line or through use of the ADDITIVE option.

DBRELATION()

DBRELATION() returns a character string indicating the link expression defined between the current work area and another. If no relationship exists, a null string ("") will be returned. The DBRELATION() function expects a numeric parameter indicating which relation should be evaluated. The syntax is:

```
<cExpression> := DBRELATION( <nRelation_number> )
```

For example,

```
use PURCHORD new
use VENDOR new index VEND_ID
select PURCHORD
set relation to VEND_ID into VENDOR

? dbrelation(1)        // Displays VEND_ID
? dbrelation(2)        // Displays ""
```

DBRSELECT()

DBRSELECT() returns the number for the work area to which the current work area is related. If no relationship exists, a zero is returned. The DBRSELECT() function expects a numeric parameter indicating which relation should be evaluated. The syntax is:

```
<nExpression> := DBRSELECT( <nRelation_number> )
```

For example,

```
select 1
use PURCHORD
select 2
use VENDOR new index VEND_ID
select PURCHORD
set relation to VEND_ID into VENDOR

? dbrelation(1)        // Displays VEND_ID
? dbrselect(1)         // Displays 1
```

DBRELATION() and DBRSELECT() can be used to temporarily remove a relatio between files and then restore it. Listing 18.7 contains a Save_rela() and Rest_rela(function to save and restore relations.

Listing 18.7 Save/restore relations

```
static rstack_:={}                         // file-wide static array

function Save_rela
aadd( rstack_,{ dbrelation(1);dbrselect(1) } )
return nil

function Rest_rela
local rsize:=len(rstack_)
local r_exp,r_area,r_work
if rsize > 0
   r_exp  := rstack_[rsize,1]
   r_area := rstack_[rsize,2]
   r_work := alias(r_area)
   dbsetrelation(r_work,;
              &("{ || "+r_exp+"} "),r_exp)
   asize(rstack_,rsize-1)
endif
return nil
```

Filtering files

A subset is a smaller set of records extracted from a larger set. For example, if a file contains all students at a school, then those students in a particular class would be a subset of the entire file. Filtering is a technique which allows you to define a set of conditions and apply it to a work area. Only records meeting the conditions will be available for operations on that work area. The filter allows you to create subsets within larger files.

SET FILTER

The SET FILTER command or the DBSETFILTER() function defines the set of conditions to be applied to a database. Each record is checked against the conditions. If the record meets the conditions in the filter statement, the record may be used by the program. If not, the program ignores the record entirely. The syntax to set a filter is:

```
// SET FILTER command

SET FILTER TO <lExpression>

// DBSETFILTER() function

DBSETFILTER( <bCondition> [, <cExpression> ]
```

<lExpression> indicates the conditions that must be met in order for the record to be recognized by Clipper. For example:

```
select STUDENTS
set filter to class == "ACCT 101"
```

<bCondition> is a code block which will be evaluated to determine if a record is included in the filter or not. It must evaluate to a logical result, **.t.** meaning include the record, and **.f.** meaning to exclude the record.

<cExpression> is a character string which will be returned by the DBFILTER() function. It is optional to the DBSETFILTER() function.

Once a filter is set, you must move the record pointer in order to activate the filter. The most common way to do so is by issuing the GO TOP command.

Most Clipper commands will recognize the filter. Any command that moves the pointer by specifying a record number will not recognize the filter. These commands include:

```
GO TO  <nRecord_number>
DBGOTO(<nRecord_number>)
```

Record movement commands are discussed later in this chapter.

SET FILTER should be used with extreme care. If the filter condition causes most records to be ignored, processing commands could spend considerable time searching the database for the next record to work on.

If the SET FILTER command is issued without a logical expression, any filter applied to the current work area is removed. You can also remove a filter by using the DBCLEARFILTER() function. Its syntax is:

```
DBCLEARFILTER()
```

DBFILTER()

The DBFILTER() function returns a string value which represents the filter condition for the current work area. If no filter has been set, a null string ("") will be returned. The syntax is:

```
<cFilter> := DBFILTER()
```

The DBFILTER() command can be used to save and restore database queries. The functions in Listing 18.8 can be used to save and restore filter conditions. The syntax is:

```
Save_filt()   // Save current filter condition
Rest_filt()   // Restore filter and apply it
```

Listing 18.8 Save/restore filters

```
static fstack_:={}                    // file-wide static array

function Save_filt
aadd( fstack_, dbfilter() )
return nil

function Rest_filt
local fsize:=len(fstack_)
local f_exp
if fsize > 0
   f_exp  := fstack_[fsize]
   dbsetfilter( &("{ ||"+f_exp+"} "), f_exp )
   asize(fstack_,fsize-1)
endif
return nil
```

Further information on working with subsets can be found in Chapter 19, which discusses the filter command in more detail as well as other approaches to working with portions of a database.

Database file status

Clipper provides functions which read the database header and return information from it. In order for these functions to work, the database must be opened in a work area.

HEADER() — size of DBF header

A .DBF file contains header information, such as record size, record count, field lists, etc., as well as the actual data. The HEADER() function returns the number of bytes in the header portion of the .DBF file. Its syntax is:

```
<nBytes> := HEADER()
```

The ***<nBytes>*** returned will be an integer length including the header and all the field definitions.

The size of the header is most often needed in determining the .DBF file's size for backup purposes.

LASTREC() — number of records in the file

The LASTREC() function reads the .DBF header and returns a numeric value indicating the last record number in the file. Its syntax is:

```
<nRecord_number> := LASTREC()
```

This number is read from the header, so it is not affected by any filter conditions. It always refers to the physical number of records in the file. Deleted records before a pack operation are included in the value returned by LASTREC().

LASTREC() is identical in operation to RECCOUNT(). Nantucket considers RECCOUNT() to be a compatibility function. It may not be supported in future compiler releases.

LUPDATE() — date file was last updated

The LUPDATE() function reads the last update year, month, and day from the header record and returns the information as a date variable. This function is useful for checking to see if the file should be backed up or if indexes need to be recreated. Its syntax is:

```
<dLast_update> := LUPDATE()
```

As an example, the code fragment below compares the file date of the index with the last update date of the .DBF file. If the file has been updated more recently than the index file's date stamp, the index file is recreated.

```
#include "DIRECTRY.CH"
local ntx_array := directory("CUST.NTX")
use CUSTOMER new
if Lupdate() > ntx_array[1,F_DATE]
   index on cust_id to cust.ntx
endif
set index to cust
```

RECSIZE() — record size

RECSIZE() sums up all the field sizes with the current work area and returns the resulting record size. If no file is opened in the current work area, RECSIZE() returns zero. The syntax is:

```
<nBytes> := RECSIZE()
```

For example, the code fragment in Listing 18.9 shows the use of RECSIZE() to determine whether or not all fields from a database can be printed on a single line.

Listing 18.9 RECSIZE() example

```
function Can_fit
if recsize() < 132
   for jj := 1 to fcount()
      ?? fieldget(jj)
   next
   ?
elseif recsize() < 255
   ?? chr(15)                  // Set printer to condensed print
   for jj := 1 to fcount()
      ?? fieldget(jj)
   next
   ? chr(18)                   // Set printer to normal printing
endif
```

The database status functions can be used in a backup utility to see if a particular .DBF file will fit on a diskette. Listing 18.10 illustrates a simple backup system using the database status functions. The syntax for Backdbf() is:

```
<logical>  := Backdbf(<cDbf_file>,<cDrive letter>)
```

Listing 18.10 Backdbf()

```
function Backdbf(cDbf_file,cDrive)
local ndrv := asc(upper(cDrive))-64
local oldfile :=trim(cDbf_file)+".DBF"
local newfile :=cDrive+":\"+oldfile
local tarr,num_recs
if header()+(recsize()*lastrec()) > diskspace(ndrv)
   go top
   do while !eof()
      num_recs := ( diskspace(ndrv)-header() ) / recsize()
      copy next num_recs to (newfile)
      ? "Insert a new diskette in drive "+cDrive
      inkey(500)
   enddo
else
   copy file (oldfile) to (newfile)
endif
return nil
```

Information about fields

The .DBF file structure allows each record to be broken into a number of fields. The field names, types, and sizes are stored in the database header. When a record from the database is read into memory, the current record is transferred into a buffer area and broken into fields. The field names are then available for use by your application program.

FCOUNT()

FCOUNT() returns an integer indicating how many fields are in the current .DBF structure. It is useful to set the upper boundary in a FOR..NEXT loop which will display all fields from the current record. Its syntax is:

```
<nFields>  := FCOUNT()
```

Listing 18.11 illustrates a simple database export utility to create a comma-delimited ASCII file from all the fields in the current work area.

Listing 18.11 Simple Export()

```
* Program...: Simple Export()
*
function Sexport(cFilename)
local nFields:=fcount()          // Determine number of fields
local jj
local oldaltfile := set(_SET_ALTFILE, cFilename)
local oldalt := set(_SET_ALTERNATE, .T.)
go top
do while !eof()
   for jj := 1 to nFields
     ?? fieldget(jj),","
   next
   ?
   skip +1
enddo
```

```
// Restore previous ALTERNATE settings
set(_SET_ALTERNATE, oldalt)
set(_SET_ALTFILE, oldaltfile)
return nil
```

FIELD()

The FIELD() function returns the field name for the specified field number. Its syntax is:

```
<cFld_name>  := FIELD( <nField_number> )
```

The FIELD() function is useful in applications where the field names may not be known. For example, Listing 18.12 illustrates a very simple generic data entry screen using the FIELD() function.

Listing 18.12 Generic data entry

```
#include "inkey.ch"
function GenDataEntry
local nFields := fcount(), jj, fld_list :={}
memvar getlist
cls
for jj:=1 to nFields
   aadd( fld_list, fieldget(jj) )
   @ jj,01 say field(jj) get fld_list[jj]
next
read
if lastkey() <> K_ESC
   for jj:=1 to nFields
     fieldput(jj,fld_list[jj])
   next
endif
return nil
```

This function can be used for very simple database structures since it offers no validation and does not check the screen positioning. It primarily demonstrates the use of the **field**() function to display a prompt where the user can enter data.

FIELDPOS()

The FIELDPOS() function returns the field number for a specified field name. Its syntax is:

```
<nField_number>  := FIELDPOS( <cField_name> )
```

The FIELDPOS() function can be used in conjunction with the FIELDGET() function to return the value of a given field when its name is known, but its position is not. For example:

```
use CUSTOMER new
? Fieldget(Fieldpos("CUST_ID"))
```

DBSTRUCT()

DBSTRUCT() will return an array which contains the field name, type, length, and decimal places for each field in the .DBF file.

The syntax for DBSTRUCT() is:

```
<Array>    := DBSTRUCT()
```

The returned **array** has four elements per field. Its structure is listed in Figure 18.2:

Figure 18.2 DBSTRUCT() array structure

Element	*Description*	*Manifest constant*
1	cField_name	DBS_NAME
2	cField_type	DBS_TYPE
3	nField_len	DBS_LEN
4	nField_dec	DBS_DEC

The manifest constants are defined in the DBSTRUCT.CH file found in the \CLIPPER5\INCLUDE directory. The structure of the returned array is the same structure that the DBCREATE() function uses to create a file. This makes it very simple to copy the structure of a .DBF file to a new file. For example:

```
the_struct := dbstruct()
dbcreate("NEWFILE.DBF",the_struct)
```

Moving about in the file

Each work area has a record pointer maintained for it. This pointer refers to a physical record number in the file. Clipper provides commands and functions to update the pointer to different record numbers. The position of the pointer determines where the next read or write operation will be performed.

SEEK — quick lookup in an indexed file

The SEEK command requires that an index file has been created and is opened for the current work area. It uses the index to determine the record number and if a match is found, positions the file at the requested record. Its syntax is:

```
SEEK <expression>
```

The DBSEEK() function can also be used to look up records in an ordered database. Its syntax is:

```
DBSEEK( <expression>, [<lSoftseek>]
```

<expression> is the key expression that the current work area's controlling index is indexed on. If the expression is found within the database, the record pointer will be moved to the appropriate record. If not, the record pointer will be positioned after the last record in the .DBF file.

<lSoftseek> is **.t.** if SOFTSEEK should be used or **.f.** otherwise. See Chapter 19 for more information on SOFTSEEK.

In addition to moving the pointer, SEEK will set a logical value indicating if a key is found or not found. This value can be determined through the FOUND() function, discussed later in this chapter.

For example:

```
use CUSTOMER new
index on upper(Cust_id) to cust.ntx
seek "NANTUCKET"
? found()                  // will return true if NANTUCKET is in
                           // the customer file, otherwise false
```

DBSEEK() can be used to seek expressions in unselected work areas. For example,

```
use CUSTOMER new index cust.ntx
use INVOICE  new
ok := CUSTOMER->( dbseek("NANTUCKET" )
```

DBSEEK() returns a logical value indicating whether or not the key was found.

LOCATE — linear search of a .DBF file

LOCATE performs a linear (top to bottom) search of the .DBF file looking for a matching condition. If the condition is found, the database is positioned at the appropriate record number and the FOUND() status is updated to true. If the data is not found, the record pointer is moved to one plus the end of the file and the FOUND() status is set to false. The syntax for LOCATE is:

```
LOCATE <scope> FOR <condition> [WHILE <condition> ]
```

The ***<scope>*** determines both the starting point and the number of records to process. The default scope is ALL, which starts at the top of the file and searches every record. The possible <scope> values are shown in Table 18.4:

Table 18.4 Scope values

ALL	Go to the top of the file and process every record.
NEXT <n>	Only process the next **<n>** records from the current position.
RECORD <n>	Only process record number **<n>**.
REST	Process all records from current record through the end of the file.

The **FOR *<condition>*** indicates what we are searching for and is a required part of the command. Once the <condition> is met, LOCATE stops and sets FOUND() to .t. If none of the records within the scope meets the condition, the LOCATE command will set FOUND() to .f.

WHILE *<conditions>* is used to narrow down the records which might be returned by the LOCATE command. The condition specified after the WHILE keyword indicates how long the search should continue. As soon as a record fails to meet the WHILE condition, the LOCATE command stops and sets FOUND() to .f.

The LOCATE command is discussed in more detail in Chapter 19, "Searching and Querying".

GO — go directly to a record number

GO is used to move directly to a record number in a database file. Its syntax is:

```
GO[TO]  <nExpression> | TOP | BOTTOM
```

The <[TO]> is optional, so GO may also be written as GOTO.

<nExpression> is a mathematical expression which is evaluated. The result of the expression determines the record number in the .DBF file to move the record pointer to.

The **TOP** keyword causes the record pointer to be moved to the first logical record. The **BOTTOM** keyword moves the pointer to the last logical record. A logical record is determined by the controlling index, and might not be the same as the physical record number.

Care should be used when positioning in .DBF files using the GO command. If the GO command specifies a record number rather than TOP or BOTTOM, any filter condition applied to the database will be ignored. It is possible to go to a record that the SET FILTER command would otherwise ignore.

There are also three functions which can be used to GOTO records in a database.

DBGOTO(). The DBGOTO() function performs the same function as the standard GOTO command. Its syntax is:

```
DBGOTO( <nRecord> )
```

<nRecord> is the record number to which the database should be positioned. DBGOTO() always returns a NIL.

DBGOTOP(). The DBGOTOP() function performs the same function as the standard GO TOP command. Its syntax is:

```
DBGOTOP()
```

DBGOTOP() moves the database pointer to the first logical record and it always returns a NIL.

DBGOBOTTOM(). The DBGOBOTTOM() function performs the same function as the standard GO BOTTOM command. Its syntax is:

```
DBGOBOTTOM()
```

DBGOBOTTOM() moves the database pointer to the last logical record and it always returns a NIL.

SKIP — skip to the next logical record

Skip increments the record pointer to advance to the next record. A parameter can be used to specify both direction and number of logical records to move. Its syntax is:

```
SKIP <nExpression>
```

<nExpression> is a mathematical expression that will evaluate to a positive or negative integer value. If the expression is positive, the record pointer will be moved that many records forward (closer to the end of the file). If the expression is negative, the record pointer will be moved backward (closer to the beginning of the file).

SKIP is most often used to move through a database and process each record. For example, the code fragment in Listing 18.13 illustrates a simple report.

Listing 18.13 SKIP example

```
use CUSTOMER new
index on upper(Cust_id) to cust.ntx
set console off
set printer on
go top
do while .not. eof()
   ? Cust_id,Cust_name,Cust_city,Cust_state
   skip +1
enddo
```

```
set printer off
set console on
```

Note: This example is intended only to illustrate the SKIP command. Refer to Chapter 15 for much more detail on printing reports.

A SKIP operation may also be performed using the DBSKIP() function. Its syntax is:

```
DBSKIP(nRecords)
```

<nRecords> is a mathematical expression that will evaluate to a positive or negative integer value. If the expression is positive, the record pointer will be moved that many records forward (closer to the end of the file). If the expression is negative, the record pointer will be moved backward (closer to the beginning of the file.)

DBSKIP() always returns a NIL.

BOF() — at beginning of work area?

The BOF() function is used to test whether the record pointer is at the beginning of the file. Its syntax is:

```
<lExpression>  := BOF()
```

The BOF() occurs when you try to move backwards from the beginning of the file. For example,

```
use CUSTOMER new
go bottom
do while !bof()              // BOF occurs when skipping
   ? CUSTOMER->Cust_name    // prior to the first record
   skip -1
enddo
```

BOF() performs a logical test rather than a physical test. If you attempt to SKIP prior to the first record in the controlling index, BOF() returns true. This may or may not be the first record in the file.

EOF() — at end of work area?

The EOF() function is used to test whether the record pointer is at the end of the file. Its syntax is:

```
<lExpression>  := EOF()
```

The EOF() occurs when you try to move forward from the end of the file. For example,

```
use CUSTOMER new
go top
do while !eof()                 // EOF occurs when skipping
   ? CUSTOMER->Cust_name        // past the last record
   skip +1
enddo
```

EOF() performs a logical test rather than a physical test. If you attempt to SKIP past the last record in the controlling index EOF() returns true. This may or may not be the last record in the file.

EOF() is frequently used to control a loop for processing all records in a database. It may also be used to check the record position after a SEEK or LOCATE. Listing 18.14 illustrates some examples.

Listing 18.14 EOF() Examples

```
local tot_due :=0,mcust
use CUSTOMER new
index on upper(Cust_id) to cust.ntx
go top
do while !eof()          // Test loop for end of file
   ? CUSTOMER->Custname,CUSTOMER->Balance
   tot_due += CUSTOMER->Balance
   skip +1
enddo

mcust := "NANTUCKET"
seek mcust
if eof()
   ? "Customer "+mcust+" not found"
endif
```

DELETED() — is current record flagged for deletion?

The DELETED() function checks the current record to see if the record has been tagged for deletion. When a record is deleted in Clipper, it is not physically removed from the file. A flag is set to indicate that this record should be considered removed from the file. The PACK command will physically remove all records with their deleted flags set to true.

The syntax for DELETED() is:

```
<lExpression> := DELETED()
```

The example below counts the total number of deleted records in a file to determine whether or not a PACK operation should be performed.

```
use CUSTOMER new
count all to del_count for deleted()
if del_count > 1
  pack
endif
```

RECNO() — current record number

The RECNO() function returns a numeric value that indicates the record number on which the work area is currently positioned. Its syntax is:

```
<nRec_number> := RECNO()
```

The record number is generally used by routines that process the .DBF by record number. For example, Listing 18.15 extracts every tenth record from the database to display upon an inventory sample report.

Listing 18.15 Inventory sample report

```
use INVENT new
go top
set console off
set printer on
? padc("INVENTORY SAMPLE REPORT",78)
do while !eof()
   if recno() / 10 == int( recno()/10 )  // every tenth record
      ? part_num, desc, on_hand, "__________"
   endif
   skip +1
enddo
set printer off
set console on
```

FOUND() — was last SEEK or LOCATE successful?

The FOUND() function is used to check the success of the last SEEK or LOCATE operation. Clipper maintains the status for each work area of the last search command. The FOUND() function reads this status and returns it as either true or false. Its syntax is:

```
<lExpression>  := FOUND()
```

The FOUND() function is most often used after a SEEK or LOCATE to determine how to proceed. For example,

```
getting := .t.
do while getting
   @ 10,10 get mcust_id picture "!!!!!!!!!!"
   read
   getting := .f.
   if lastkey() <> 27
       seek mcust_id
       if found()
          Edit_cust()
       else
          @ Maxrow(),02 say "CUSTOMER not found, try again."
          getting := (inkey(500) <> 27)
       endif
   endif
enddo
```

Extracting data from the file

Once the record pointer is positioned, the fields in the .DBF file are available for use. Clipper transfers the data from the record in the .DBF to a buffer area established for each work area. Fields may be extracted from this buffer area by using the field name as any other variable. Clipper will translate the request for a field into an appropriate position, size, and type within the buffer and return that data.

Assignment operator

The simplest way to extract data from a work area buffer is through the assignment operator. The syntax is:

```
<memory_variable>  := <cField_name>
```

The field may also be treated just like a regular memory variable. It is important to note that if the field is treated like a memory variable, data in the .DBF file can be updated.

A field name and a memory variable might have the same name. If this is the case, the field will normally take precedence. It is a good habit to precede all field names with the alias identifier. For example:

```
? CUSTOMER->Id_code   // displays ID_CODE from CUSTOMER.DBF
```

In addition, Clipper offers two statements to direct the compiler how to interpret ambiguous variable names. These are the **field** and **memvar** statements.

FIELD. The field statement declares a list of variables which should be considered field names if they are not specifically identified. Its syntax is:

```
FIELD <cField_list>  [IN <cAlias>]
```

For example, the code fragment below causes the compiler to consider CUST_CODE, NAME, and ADDRESS to be fields rather than memory variables. VEND_CODE,VNAME, and VADDRESS are established to be fields from the VENDOR database file.

```
FIELD cust_code,name,address
FIELD vend_code,vname,vaddress IN vendor
```

If a field statement is used, it must precede all executable statements within the function or procedure. The **field** scope is the current procedure. The **field** statement may be declared before any functions or procedures within a .PRG file. If the .PRG file is compiled using the **/n** switch, the **field** declarations have a scope of the entire program file.

The **field** statement also allows the compiler to check for undeclared variables when you use the **/w** switch.

MEMVAR. The **memvar** statement declares a list of variables which should be considered memory variable names if they are not specifically identified. Its syntax is:

```
MEMVAR <cField_list>
```

For example, the code fragment below causes the compiler to consider NAME, ADDRESS,CITY, and STATE to be memory variables rather than fields.

```
MEMVAR name,address,city,state
use EMPLOYEE new
? name             // Assumed to be a memory variable
? EMPLOYEE->Name   // A field from the employee database
```

If a **memvar** statement is used, it must precede all executable statements within the function or procedure. The **memvar** scope is the current procedure. You can change the **memvar** scope to the entire .PRG file by declaring it before any function or procedure name and compiling with the **/n** switch.

The **memvar** statement also allows the compiler to check for undeclared variables when you use the **/w** switch.

FIELDGET()

The FIELDGET() function retrieves a field value from the current record for a specified field number. Its syntax is:

```
<value> := FIELDGET( <nField_no> )
```

The FIELDGET() function is very useful for extracting data from databases where the field name is not known. This allows for generic, data-independent functions to be created.

Here's a scatter function which transfers all the fields from the current record into an array.

```
function Scatter( nRec_num )
local numfld:=fcount(), jj, retarry_:={}
nRec_num := if(nRec_num==nil, Recno(),nRec_num)
goto nRec_num
for jj := 1 to numfld
   aadd(retarry_,fieldget(jj))
next
return retarry_
```

Putting data into the file

Information from Clipper's memory can be written into a .DBF file in several ways. The information will be transferred from a memory variable into the work area's buffer. Once the .DBF file pointer moves from that record, the buffer will be written back to the disk.

REPLACE

The REPLACE command causes the values of individual fields to be updated in the database. Its syntax is:

```
REPLACE <field_name> WITH <expression> [<scope>]
           [FOR <lcondition> WHILE <lcondition>]
```

<field_name> is the name of the field from the work area that should be updated. If the field name is unknown, the FIELDPUT() function should be used rather than macro-expanding the field name.

<expression> indicates the value that should be written into the work area buffer for a particular field. Its type must agree with the declared type of the field, or a data type mismatch run-time error will occur. The expression may be a constant, a variable, a field from another work area (preceded by that work area's alias), or some combination of these.

<scope> determines both the starting point and the number of records to replace. The default scope is the current record only. Note that this is different from the default scope on other commands, such as LOCATE, SUM, and AVERAGE. On those commands the scope defaults to ALL. The possible <scope> values are shown in Table 18.4.

The **FOR *<condition>*** indicates which records should be updated with the expression. If the record meets the condition, the specified fields will be updated. If the record does not meet the specified condition, the fields in that record will not be changed. Regardless of whether the field is updated, the REPLACE command continues until the scope is completed.

The **WHILE *<condition>*** indicates the condition that must be met in order for the REPLACE command to continue processing. As long as the condition is satisfied, the specified fields will be updated. Once a record does not meet the specified condition, the REPLACE operation stops. The WHILE condition, if specified, overrides the scope.

Listing 18.16 illustrates several examples of the REPLACE command.

Listing 18.16 REPLACE command examples

```
use EMPLOYEE new
index on upper(Name) to employ
seek "JOE BOOTH"
if found()
   // Give the employee a 5% salary increase
   replace salary with salary*1.05
endif
// Give all employees a cost of living raise
replace all salary with salary *1.02
// All CA employees are being transferred on Dec. 15, 1991
replace all trans_date with ctod("12/15/91") ;
      for EMPLOYEE->State  == "CA"
```

FIELDPUT()

This function allows you to replace field contents without resorting to the use of macros to expand to the field name. The syntax is:

```
FIELDPUT( <nField_no>,<expression> )
```

<nField_no> indicates which field number should be replaced. The expression indicates the value that should be replaced into that field. If the type of the expression does not match the field type, a type mismatch run-time error will occur.

Listing 18.17 illustrates a GATHER user-defined function which will write array elements back to the .DBF file. Its syntax is:

```
Gather( <nRecord_number>,<aField_array> )
```

Listing 18.17 Gather

```
function Gather( nRec_num,retarry_ )
local numfld:=fcount(),jj
nRec_num := if(nRec_num==nil,Recno(),nRec_num)
goto nRec_num
for jj := 1 to numfld
   fieldput( jj,retarry_ )
next
return nil
```

Adding data to the file

Clipper can add a single record to the file by adding a record filled with spaces or it can add a number of records from other sources.

Adding records to the file

The APPEND BLANK command or the DBAPPEND() function can be used to add new records to a database file. The syntax is:

```
// APPEND command

APPEND BLANK

// DBAPPEND() function

DBAPPEND()
```

When the APPEND BLANK command or DBAPPEND() function is performed, a new empty record is added to the end of the file and the record pointer is positioned to this record.

For example, Listing 18.18 illustrates a generic database update program whi either adds a new record or updates the current record depending upon the paramet passed.

Listing 18.18 Append blank example

```
function Cust_edit( nRecord,fld_array )
local jj, fld_count := fcount()
nRecord := if( nRecord==nil,0,nRecord )
if nRecord == 0
   append blank
else
   goto nRecord
endif
for jj:=1 to fld_count
  Fieldput(jj,fld_array[jj])
next
return nil
```

Removing data from the file

Each record in a file has a flag indicating whether or not the record has been deleted. Clipper provides commands and functions to manipulate that flag.

Flagging records for deletion

The current record may be flagged for deletion using either the DELETE command or the DBDELETE() function. The syntax is:

```
// DELETE command

DELETE

//  DBDELETE() function

DBDELETE()
```

The current record will be flagged for deletion. The file will still be in the database until a PACK command physically removes it.

Recalling records from deletion

If the current record is flagged for deletion, the flag may be removed using either the RECALL command or the DBRECALL() function. The syntax is:

```
// RECALL command

RECALL

//  DBRECALL() function

DBRECALL()
```

The current record's deletion flag will be turned off. This will prevent a PACK operation from removing the record.

Physically removing the records

At some point, your application may need to reclaim the space being used by deleted records. This is done by issuing the PACK command. Its syntax is:

```
PACK
```

Pack operates on the current work area and makes a new copy of that work area, without deleted records. Once a pack is performed, the physical size of the file is reduced (if records were removed) and the records are gone.

Closing the file

After you have completed all updates of a database, the file should be closed. This can be accomplished in Clipper 5 using the USE command or the DBCLOSEAREA() function. The syntax to close a work area is:

```
// USE command

USE

// DBCLOSEAREA() function

DBCLOSEAREA()
```

The currently selected work area will be closed. All pending updates are written to the disk and all locks are released. Be sure to select the appropriate work area to close before issuing a USE command or a DBCLOSEAREA() function call.

If you need to close all work areas at once, you may use either the CLOSE DATABASES command or the DBCLOSEALL() function. The syntax is:

```
// CLOSE DATABASES command

CLOSE DATABASES

// DBCLOSEALL() function

DBCLOSEALL()
```

Both the DBCLOSEAREA() and the DBCLOSEALL() functions return a NIL.

Reducing storage needs

Data compression techniques reduce the amount of storage space required for a data item. **Date** types can be reduced to two bytes of storage and **numeric** types can have their storage requirements cut in half. In this section, we will explore techniques to accomplish data compression and provide some UDFs that can be used to reduce disk storage requirements.

Compressing numeric information

To perform numeric compression, we need to view the number as a string of digits, rather than a numeric value. For example, the number 778,242 should be viewed as 6 one-character fields, the first character field being "7", the second field "7", the third field "8", and so on. The possible values for any of these fields can be "0" thru "9", which means only ten different values are possible per character. If we combine two fields together, we have a possibility of 100 different values. Now we use the ASCII chart (see Appendix A) to our advantage. (CHR(0) on the ASCII chart is used to terminate strings and shouldn't be used in this type of application.) The ASCII chart has 255 possibilities which can be represented by a single character, and Clipper has built in functions (CHR and ASC) which understand the ASCII chart. Now let's apply it to our advantage with the above string of digits.

First, we will break the string into groups of two digits each so that 778,242 can be viewed as follows

"77" "82" "42"

Now we use the ASCII chart, and get the character that the ASCII chart has for each of the above numbers, so that the original number can be represented as

CHR(77) = M   CHR(82) = R and CHR(42) = "*"

So the string "MR*", which is only three characters in length, contains the ASCII codes 77 82 42. Therein lies our compression ability, taking the ASCII equivalent character for each two digits in the number. By applying this technique, all numeric fields can be reduced by 50%. The functions in Listing 18.19 provide the capability to compress and uncompress a numeric field. The syntax for the numeric compression functions are:

```
<cCompressed>  := Sqz_n( <nValue>,<nSize> [,<nDecimals>] )
<nValue> := Unsqz_n( <cCompressed>,<nSize> [,<nDecimals>] )
```

The *<nSize>* must be larger than the size of the string you are passing to the SQZ_N() function.

Listing 18.19 Sqz functions

```
function Sqz_n(_number,_size,_dec)
local tmpstr,retstring:="",k
_dec      := if(_dec==nil,0,_dec )
_number   := _number * (10**_dec)
_size     := if(_size/2<>int(_size/2),_size+1,_size)
tmpstr    := str( _number,_size)
tmpstr    := strtran(tmpstr," ","0")
for k := 1 to len(tmpstr) step 2
   retstring:=retstring+chr(val(substr(tmpstr,k,2))+100)
next
return retstring

function Unsqz_n(_compressed,_size,_dec)
local _tmp:="",k,multi,answer
_size:= if(_size/2<>INT(_size/2),_size+1,_size)
for k:=1 to len(_compressed)
   _tmp:=_tmp+str(asc(substr(_compressed,k,1))-100,2)
next
_tmp   := strtran(_tmp," ","0")
answer := substr(_tmp,1,_size-_dec)+"."+;
          substr(_tmp,_size-_dec+1)
return val(answer)
```

The SQZ functions allow a numeric value to be compressed when stored in the database, and to be uncompressed when retrieved. For example, using the SQZ functions, a REPLACE statement might be coded:

```
replace TRANS->Cust_id with Sqz_n(mcust_id,8),;
        TRANS->Amount  with Sqz_n(mamount,12,2)
```

and to retrieve the information from the file, the code might be written as:

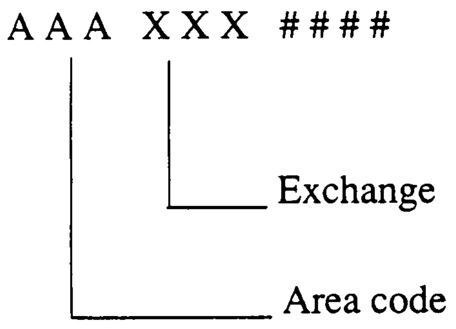
```
mcust_id := Unsqz_n(TRANS->Cust_id,8,0)
mamount  := Unsqz_n(TRANS->Amount,12,2)
```

Compressing phone numbers

A phone number, including the area code, may be compressed from ten digits to a four-character string. This represents a savings of 60% on disk space.

To understand how to compress a phone number, we need a little background on area codes and phone numbers. A phone number consists of three pieces. These are shown in Figure 18.3.

Figure 18.3 The Components of a phone number



To dial a number within your area code, you dial seven digits. For a number in a different area code, you dial ten digits: the three digit area code, followed by the seven digits of the number. The structure of phone numbers is such that the middle digit of all area codes must be a zero or a one. Originally, the middle digit of exchanges was always above one, but some area codes have gotten so crowded that the number one appears in some exchanges.

Once we realize that the middle digit of the area code must be a zero or a one, we can reduce the number of possible phone numbers some. After removing punctuation from the phone number, we are left with ten digits. These ten digits can range from zero through 9,999,999,999. However, if we transpose the first and second digits, we reduce the range of possible phone numbers to zero through 1,999,999,999.

Once the range is reduced, we can use the L2BIN() function to store the number. L2BIN() has a range of zero to 4,294,987,295.

This range is more than sufficient for the storage of phone numbers. The function Sqz_p() in Listing 18.20 takes a phone number and area code and compresses it to a binary string. Unsqz_p(), in Listing 18.22, takes the binary string and returns a punctuated phone number. The syntax for Sqz_p() is:

```
<cBinary_string> := Sqz_p( <cPhone_number> )
```

The syntax for Unsqz_p() is

```
<cPhone_numbr> := Unsqz_p( <cBinary_string> )
```

Listing 18.20 Sqz_p

```
function Sqz_p(cPhone)
local cBinary,cHold
cHold := strtran(cPhone,"(","") // Remove various
cHold := strtran(cHold ,")","") // punctuation characters
cHold := strtran(cHold ,"-","") // ( ) - and space
cHold := strtran(cHold ," ","")
cHold := substr(cHold,2,1)+substr(cHold,1,1)+;
               substr(cHold,3,8)
cBinary := L2bin( val(cHold) )
return cBinary
```

Listing 18.21 UnSqz_p

```
function Unsqz_p(cBinary)
local cPhone,nHold,cHold
nHold  :=Bin2l(cBinary)
cHold  :=Alltrim(str(nhold,11))
cHold  :=if(len(cHold)<>10,"0"+cHold,cHold)
cPhone :="("+substr(cHold,2,1)+substr(cHold,1,1)+;
              substr(cHold,3,1)+") "+substr(cHold,4,3)+;
              "-"+substr(cHold,7,4)
return cPhone
```

Compressing dates

Dates may also be compressed to save room in your .DBFs. The eight-byte date field can be compressed into a two-byte character field. To perform this compression, we must first explore some tricks of the Clipper compiler. Let's take the date January 22, 1989. Stored as a date, this field would contain 01/22/89. Another way of saying the same date is that 01/22/89 is 3,309 days after 01/01/80 (our base date). This is where date compression can occur.

To compress a date, we first convert it to an integer number of days past some base date. We need to choose a base date that we are not going to precede. For a transaction system, 01/01/80 is probably a good choice. Since an integer can be a number from 0 through 65,535, this allows us approximately 179 years worth of dates starting at 01/01/80. If you don't plan to be coding in the year 2159, this range should be acceptable. If however, you are coding a system that needs a larger range of dates, you could change the function to use the long integer format which would allow for 2,147,418,112 days or over five million years.

Once the date is expressed as an integer number of days, we can use the Clipper functions of I2BIN() and BIN2W() to reduce the storage requirements of the integer. I2BIN() takes an integer parameter and returns a two-character binary representation of that integer. BIN2W() does the reverse, taking the two characters and returning an integer.

The Sqz_d() and Unsqz_d() functions below provide compress and uncompress dates using the 179-year window. The syntax to compress the date is:

```
<cBinary_string> := Sqz_d( <dSome_date> )
```

and to restore the date from the compressed string:

```
<dSome_date> := Unsqz_d( <cBinary_string> )

#define BASE_DATE ctod("01/01/1980")

function Sqz_d(dSquish)
return I2bin( dSquish-BASE_DATE )

function Unsqz_d(cBinary)
return BASE_DATE + Bin2w(cBinary)
```

If 179 years is not enough of a date window, then the following versions of these functions can be used to compress dates down to four characters and yet contain any date that Clipper accepts.

```
#define BASE_DATE ctod("01/01/1980")

function Sqz_d(dSquish)
return L2bin( dSquish-BASE_DATE )

function Unsqz_d(cBinary)
return BASE_DATE + Bin2l(cBinary)
```

The BASE_ DATE value is the earliest date that will be accurately returned. Change the define setting if you need to work with dates earlier than January 1, 1980.

Compressing time

While Clipper does not specifically provide a time data type, it does provide a function, (TIME()) which will return the system time.

The format for a time string is HH:MM:SS. In its formatted form, the time string requires eight characters of storage. If the seconds are not needed, a time string can be compressed to two bytes. If seconds are needed, we can compress the string to three bytes.

To compress time, we first must remove the colon characters which separate the components. After they are removed, we are left with a string of six numbers. If seconds are not needed, the string is only four numbers. By using the numeric compression user-defined functions, we can reduce the storage requirement of a time string. The Sqz_t() function in Listing 18.22 takes a time string and returns a compressed string from it. Its syntax is:

```
<cCompressed_string> := Sqz_t( <cTime>,<lSeconds> )
```

The syntax for Unsqz_t() is

```
<cTime> := Unsqz_t( <cCompressed_string> )
```

The Unsqz_t() function determines whether or not seconds are needed by the size of the string passed to it.

Listing 18.22 Sqz_t

```
function Sqz_t(cTime,lSeconds)
local cHold,nHold
lSeconds :=if(lSeconds==nil,.f., lSeconds)
cHold := strtran(cTime, ":", "")   // Remove colons
nHold := val(cHold)                // Convert to numeric
return if(lSeconds, Sqz_n(nHold,6,0), Sqz_n(nHold,4,0))
*
```

```
function Unsqz_t(cComp)
local nSize := len(cComp) * 2
local cHold := Unsqz_n(cComp, nSize, 0)
cHold := str(cHold, nSize, 0)
cHold := if(nSize=4, cHold+"00", cHold)
return substr(cHold,1,2)+":"+;
       substr(cHold,3,2)+":"+;
       substr(cHold,5,2)
```

Summary

After reading this chapter you should be familiar with the primary database file structure that Clipper uses. You should know how to create index files and how to move about the file. Getting data from the file and updating information should also be clear. Finally, if disk space is at a premium, you should be able to use the compression techniques and functions to reduce your storage requirements.

CHAPTER 19

Searching and Querying

Searching is the process of trying to determine which record or records meet the search criteria. Querying is the process of determining information about the entire database or a subset of it. For example, a search would indicate which record contained the name Smith while a query might count how many Smiths appear in the database.

In this chapter we will explore how Clipper can be used to search the database, both with and without the benefit of index files. We will also discuss some ways to expand searching techniques. Finally, the concept of a file subset will be described, as well as Clipper techniques for creating one.

Locating your data

The easiest method to find a particular record is to perform a linear search. The primary benefit of a linear search is that the database order is unimportant. The drawback is that of speed, or lack thereof. A linear search starts at the top of the file and checks every record until a match is found. For a small database, under 100 records, this process will be performed fairly quickly. If the database is large, the linear search technique should be used only if no other options are available.

LOCATE

Clipper implements a linear search using the LOCATE command. Its syntax is:

```
LOCATE <scope> FOR <condition> [ WHILE <condition> ]
```

The ***<scope>*** determines both the starting point and the number of records to process. The default scope is ALL, which starts at the top of the file and searches every record. The possible <scope> values are shown in Table 19.1:

Table 19.1 Scope values

ALL	Go to the top of the file and process every record.
NEXT <n>	Only process the next <n> records from the current position.
RECORD <n>	Only process record number <n>.
REST	Process all records from current record through the end of the file.

The ***FOR <condition>*** indicates what we are searching for and is a required part of the command. Once the <condition> is met, LOCATE stops and sets FOUND() to true (**.t.**) If none of the records within the scope meets the condition, the LOCATE command will set FOUND() to false (**.f.**).

The ***WHILE <condition>*** is used to narrow down the records which might be returned by the LOCATE command. The condition specified after the WHILE keyword indicates how long the search should continue. As soon as a record fails to meet the WHILE condition, the LOCATE command stops and sets FOUND() to .f. Let's look at some examples:

Assume we have a database file listing all sales representatives in each state for all states. The structure of the file is shown in Figure 19.1.

Figure 19.1 Sales Rep structure

Name	*Type*	*Length*	*Decimals*
State	Char	2	0
Sales_rep	Char	3	0
Ytd_sales	Numeric	12	2

To locate the first sales rep in California, we would use

```
locate all for State == “CA”
```

If the database is ordered by state, we could use:

```
locate all for State == "CA"      // Find first CA salesman
     locate rest for Ytd_sales > 5000 while State == "CA"
```

These commands would cause the first salesman with year-to-date sales of over $5,000 to be found.

The LOCATE command can be continued from the current point by either the CONTINUE command or by changing the scope of the command.

CONTINUE

The CONTINUE command causes the LOCATE command to search for the next record which meets the condition. Its syntax is simply:

```
CONTINUE
```

For example, Listing 19.1 illustrates a function which uses a loop to list all California sales representatives within the .DBF file.

Listing 19.1 LOCATE/CONTINUE loop

```
use SALESREP new
locate all for SALESREP->State == "CA"
do while found()
   ? SALESREP->Sales_rep, SALESREP->Ytd_sales
   continue
enddo
```

One problem with CONTINUE however, is that it does not maintain the **scope** nor the **while** condition of the previous LOCATE. This basically restricts the use of CONTINUE to searches where the scope is ALL and no WHILE condition is used.

Changing scope

If the CONTINUE command cannot be used, another LOCATE command should be issued with a scope of REST and the WHILE condition specified. Listing 19.2 illustrates how the California sales reps with year-to-date sales over $5,000 could be listed if the .DBF file were ordered by state.

Listing 19.2 LOCATE loop

```
use SALESREP new
locate all for State == "CA" .and. ;
     Ytd_sales > 5000                // First CA sales rep
do while found()
   ? SALESREP->Sales_rep, SALESREP->Ytd_sales
   locate rest for Ytd_sales > 5000 while State == "CA"
enddo
```

The use of the LOCATE command allows you to search a .DBF file regardless of how it is sorted without requiring an index file. The speed of performance however, is directly related to the number of records Clipper can read into memory at one time.

For databases with many records but small record sizes or with a small number of records, LOCATE may be fast enough. For larger databases, indexes and/or sorts would probably be better.

Finding it more quickly

You can speed up searches dramatically if the database is in order by the key you are looking for. Clipper provides two ways to order the database by keys: sorting and indexing. Sorting physically rearranges the records in the database while indexing prepares a second file, called the index, which logically orders the record in the database. The functions available for searching depend upon how the database is ordered.

Sorted databases

You can create a new sorted database file from an existing .DBF by using the Clipper SORT command. Its syntax is:

```
SORT TO <cDbf> ON <cField>/<opts> ;
     <scope> WHILE <cond> FOR <cond>
```

The sort command creates a new .DBF file copied from the current work area. The new .DBF is physically sorted in the specified order.

<cDbf> specifies the name of the new file to create. If no extension is specified, it will default to .DBF.

<cField> indicates the fields that the new file should be sorted on. More than one field may be specified. The first field is the primary sort key, the second field specified is the secondary key, and so on. Clipper supports any number of sort fields.

Normally the fields will be sorted in ascending case-sensitive sequence. The *<opts>* can be specified to change the default. Valid options are listed in Table 19.2.

Table 19.2 Sort field options

A - Ascending order (this is the default).
D - Descending order.
C - Case insensitive, known as dictionary order.

The options are specified by putting them immediately following the field name, separated by a slash character (/). The A and D options cannot both be specified or Clipper will produce a syntax error. You can put two options together after the slash if need be. Only one slash should be used regardless of the number of options.

At least one field must be specified for the SORT command to operate. If multiple fields are used, it is possible to have one field descending and another field ascending.

The ***<scope>*** indicates which records should be sorted. The <scope> is optional and defaults to ALL records. If specified, any scope listed in Table 19.1 can be used.

The ***WHILE <cond>*** is an optional clause which can be used to restrict the records included in the sort operation. Only records starting at the current record and ending when the condition fails will be included in the sorted database.

The ***FOR <cond>*** is also an optional clause which can be used to specify the records which get included in the sort operation. Only records meeting the condition will be included. If a record does not meet the condition, that record will not be included. Sort performs much of its work in memory and then writes it to a temporary disk file. The SORT operation uses at least three file handles (one each for the original .DBF, the new .DBF, and the temporary file). If the original .DBF file contains a memo field, at least two more handles will be required. A DOS error 4 on a SORT operation indicates not enough handles are available to perform the sort.

If the file being sorted is empty, a run-time error will occur when attempting to sort the file. You should include a test to establish that at least one record exists before you perform the SORT command.

Using the sorted file

Once the SORT operation is complete, you have a second .DBF file. You could erase the old file and rename the sorted version to take advantage of the speed increase offered by using a sorted file. For example:

```
use CUSTOMER new
if lastrec() > 0
   sort to New_cust on id_code/A all
   if file("New_cust.dbf")
      close databases
      rename CUSTOMER.DBF to CUSTOMER.OLD
      rename NEW_CUST.DBF to CUSTOMER.DBF
   endif
endif
use
use CUSTOMER new
```

It is important to realize that the sorted file will quickly become unsorted if you append records into the file. A sorted database should be used when you expect very little update activity to occur in the file. In one particular application file handles were at a premium and not available for indexes. The system used two files, a sorted master listing and a daily transaction log. Lookups were performed first against the sorted master using the Bsearch() UDF from Listing 19.3. If that failed, the daily transaction file was searched using LOCATE. At the end of the day, the two files were merged and a sorted database was produced for the next day.

Restricting LOCATE

If the database is sorted on the key you are searching for, you can speed the function of the LOCATE command by restricting the records it checks by using the WHILE option. For example, assuming the database is sorted by Sales. The command

```
locate all for Sales == 1500
```

will search the entire database. However, since the database is sorted, we can restrict the LOCATE command by adding the WHILE clause. For example:

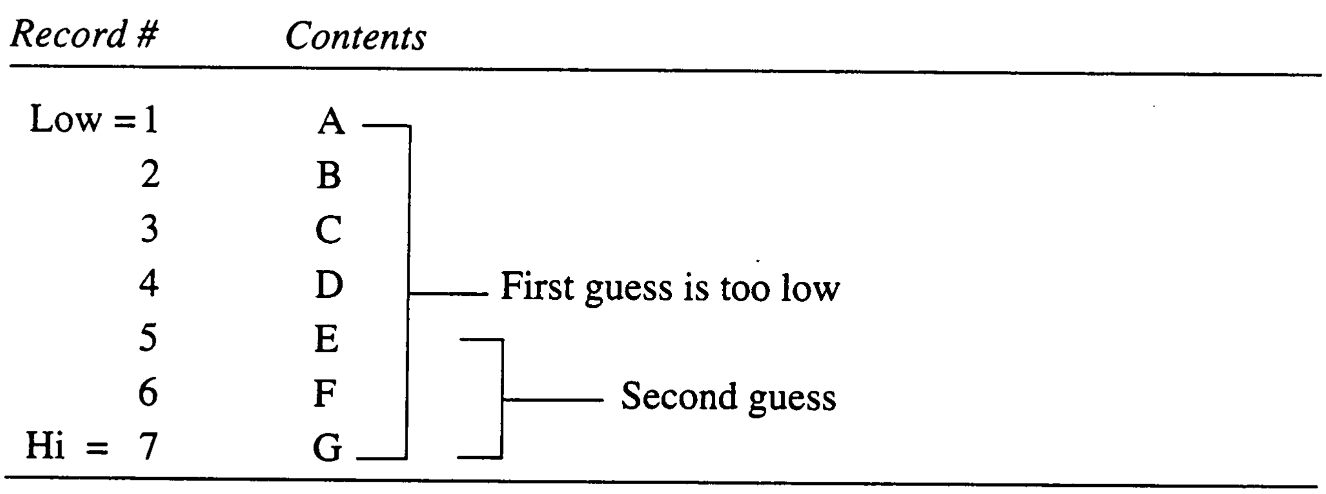
```
locate all for Sales == 1500 while Sales <= 1500
```

This command tells the LOCATE command to stop searching if the Sales field has a value above 1500. Since the database is sorted by sales, we know that once the sales amount exceeds 1500, it is impossible to find a sales value equal to 1500.

Bsearch()

Another technique to use with a sorted database is the binary find. The binary search works by cutting the search space in half repeatedly until it either finds a match or fails. Figure 19.2 illustrates how a binary search would work to find the letter F:

Figure 19.2 Binary search



First guess is four :	low + high (1+7) divided by two
Second guess is six :	new low = midpoint (four) +one or five
	same high of seven
	new midpoint is (low+high (5+7) / 2)

The Bsearch() user-defined function in Listing 19.3 can be used to perform a binary search on a sorted database. Its syntax is:

```
Bsearch( <field>,<key> )
```

where ***<field>*** is the numeric field number to be searched and ***<key>*** is the value to search for. If the search is successful, the database will be positioned on the proper record and the function will return true. If the search fails, a false value will be returned.

Listing 19.3 Bsearch()

```
function Bsearch( nfield,the_key )
local retval:=.f., bottom:=Lastrec(), top:=1, mid:=0
do while top <= bottom .and. !retval
   mid := int( (top+bottom)/2 )
   goto mid
   if fieldget(nfield) == the_key
      retval :=.t.
   elseif fieldget(nfield) < the_key
      bottom := mid +1
   else
      top := mid -1
   endif
enddo
return retval
```

Indexed databases

A database file may also be indexed to provide quick access to the information in the file. The index is a separate file which must be opened and updated whenever the .DBF file is opened. See Chapter 18 for more information on opening and working with index files.

An index is created by reading each record in the file and generating a key for it. That key and the record number are stored in a binary tree for rapid retrieval. An index file is created by selecting the work area and specifying the expression that should be used. The syntax to create an index is:

```
INDEX ON <expression> TO <cFilename>
```

If a database has more than one index, only one index is the controlling index at any one time. This index determines the expression that SEEK and FIND operate against. Each index is numbered ordinally by its position on the SET INDEX TO command. The default controlling index is the first one. The SET ORDER TO command can be used to switch the controlling index to another index in the list. The syntax for SET ORDER TO is:

```
SET ORDER TO <nOrder>
```

The **<nOrder>** must be between one and the number of indexes opened with the .DBF file. The order can be zero, which causes the database to be viewed in record number order. Regardless of the index order, all open indexes will be updated when the database is written to.

Case considerations

When indexing on character data it is important to keep in mind that indexes are case-sensitive. You should create your index using either UPPER() or LOWER() and perform the lookup based upon the proper case.

For example:

```
index on upper(emp_name) to employee
mname := "McMaster"
seek upper(mname)
```

By using the UPPER() or LOWER() function in the index expression, you can prevent the end-user from not finding his data because it was entered in a different case than the one in which it is stored.

Indexing on dates

If a date field is to be used in an index, the DTOS() function can convert the date into a string format that will properly list dates in chronological order. The format of DTOS() is YYYYMMDD. This format stays the same regardless of the setting of the SET DATE command.

For example:

```
index on dtos(hire_date) to seniority
mhired := ctod("01/22/89")
seek dtos(mhired)
```

Use of the DTOS() function is particularly important if the index is being used to display the data in date order.

DESCEND()

Indexes are normally created in ascending order. You can create a descending index by using the DESCEND() function. DESCEND() takes a parameter of any type and returns its complemented form. For example,

```
? Descend("Clipper")   // produces  ╝öùÉÉ¢Ä
? Descend(1991)        // produces  -1991.00
```

To create a descending index use:

```
INDEX ON DESCEND( <expression> ) TO <ntx_file>
```

Since DESCEND() is in EXTEND.LIB rather than CLIPPER.LIB, you will need to declare the function as external if you do not explicitly use it in your code. This would occur if DESCEND() was used in an index expression which was created by a different application, but had to be updated from within your application.

DESCEND() and its funny-looking return values are discussed in more detail in Chapter 18.

SEEK

The SEEK command is used to search an index for a particular key. SEEK compares its parameters with all entries in the controlling index file. If a match is found, FOUND() is set to true and EOF() is set to false. The database is positioned at the found record. The syntax for SEEK is:

```
SEEK <expression>
```

The ***<expression>*** must agree with the expression that the database is currently indexed on. If the type of the expression does not agree with the type of the index, a type mismatch run-time error will occur. If the type agrees, but the expression is not the same, Clipper will SEEK the key anyway. The results probably will not be the key you are looking for.

FIND

The FIND command works just like the SEEK command with a small exception. The argument to the FIND command does not require quotation marks and is considered a literal value rather than a memory variable. For example:

```
mvar := "Smith"
seek mvar          // Looks for Smith
find mvar          // Looks for mvar
find &mvar.        // Looks for Smith
```

The syntax for the FIND command is:

```
FIND <key_to_find>
```

If the key is found, FOUND() is set to .t. and EOF() is set to .f. In addition, the database is positioned on the FOUND() record.

The FIND command was designed with an interpretive environment in mind. For a user at a dot-prompt, it is more intuitive to specify the parameter without the quotation characters. Since Clipper provides no interpretive environment, FIND is considered a compatibility command and should be replaced by SEEK.

Continuing a SEEK or FIND command

It is important to understand that Clipper enforces the logical order imposed by the index on the file. Once a key is found, the way to determine the next record with the same key is by issuing a SKIP command.

For example, if the SALESREP database was indexed on state, the loop to list all salesmen in California could be coded as shown in Listing 19.4.

Listing 19.4 SEEK loop

```
use SALESREP index STATE new
seek "CA"
do while !eof() .and. State == "CA"
   ? SALESREP->Sales_rep, SALESREP->Ytd_sales
   skip +1
enddo
```

Since the index imposes a logical view, i.e. state order, on the database, the records will be read from the .DBF file in that order. Therefore the SKIP command will move to the next logical record, which might not be the next physical record in the file.

Expanding search capabilities

While the LOCATE and SEEK commands can be used to provide fast lookups of data, they both require the user to know the key they are looking up. For many applications, the user might not know the key value and will need some assistance in finding the proper record. Clipper provides commands which can be used to expand the searching capabilities of the language.

Partial searches

Many times the user knows only the first few characters of the data being sought. A partial search would allow the user to enter the characters he knew and would show a window of records from which the user could choose. Clipper supports partial lookups based on the first characters of the index expression. If a key is sought which is smaller in size than the index key, Clipper will find the first key beginning with those letters. Using a loop and array we could quickly build a picklist for the user to select the record from. Partfind() in Listing 19.5 provides a function to perform a partial search. Its syntax is:

```
<logical>  := Partfind( <seek_expr>,<field_no> )
```

The function returns a logical true if a record was selected and a false if not. It will also position the database to the found record.

Listing 19.5 Partfind()

```
function Partfind(expSeek,field_no)
local x:=0, plist:={}, y:=len(trim(expSeek)), rec:={}
seek trim(expSeek)
```

```
if found()
   do while !eof() .and. ;
      trim(expSeek)==substr(fieldget(field_no),1,y)
      Aadd(plist,fieldget(field_no))
      Aadd(rec,recno())
      skip +1
   enddo
   x :=achoice(5,20,15,45,plist,.t.)
   if !empty(x)
      goto rec[x]
      return .t.
   endif
endif
return .f.
```

SOFTSEEK

Clipper allows you to control the position of the record pointer after a SEEK command fails. Normally, if a SEEK command fails, the record pointer is positioned to the last record plus one. The EOF() function will return true and the FOUND() function returns false.

The **SET SOFTSEEK** command can be used to change the way the record pointer is positioned. If SOFTSEEK is set to ON and a SEEK fails, FOUND() will still return false. EOF(), however, will return false also. This indicates that the database is not positioned at the end of the file but rather is positioned at the next highest key in the .DBF file.

Figure 19.3 illustrates how SOFTSEEK operates:

Figure 19.3 Softseek

Record	*Key value*	*SEEK "Kelly"*
1	Booth	
2	Forcier	FOUND() = .f. / EOF() = .f.
3	Lief	Record number is three
4	Uchman	(Next highest key past Kelly)
5	Yellick	

The syntax for SET SOFTSEEK is:

```
SET SOFTSEEK [ ON | OFF | <logical> ]
```

The default value is OFF. The proximity search in Listing 19.6 uses SOFTSEEK to find records close to the unknown key.

Proximity searches

A proximity search is used to search a database for a key value and return a subset of records on either side of the key. It allows a user who does not know the key he is looking for to see keys that are close. He can then choose a key from a list. For example, suppose a customer remembers making a deposit in his bank sometime in the first couple of weeks in July, but does not remember the exact date. To perform a proximity search, the teller might enter July 7 and 7 days. The system would respond by showing a list of all transactions between July 1 and July 14. It does not matter whether a transaction occurred on July 7 or not.

Listing 19.6 contains a function called Psearch() which implements a proximity search feature. Its syntax is:

```
<array> := Psearch( <key>,<nRange>,<nField> )
```

The array returned is a list of record numbers which indicate the range of records around the specified key.

Listing 19.6 Psearch()

```
function Psearch(cKey,nRange,nField)
local arr_:={}
local oldsoft := set(_SET_SOFTSEEK, .T.)
seek cKey
skip -nRange
```

```
do while len(arr_) < (nRange*2) .and. !eof()
   aadd(arr_,fieldget(nfield))
   skip +1
enddo
set(_SET_SOFTSEEK,oldsoft)  //restore previous SOFTSEEK setting
return arr_
```

SOUNDEX()

The SOUNDEX() function returns a code that represents a phonetic spelling of a name. It is an attempt to allow names with similar sounds but different spellings to be grouped together. This increases the likelihood of finding a name even if the user is unsure of the spelling.

SOUNDEX() works by building a 4-character key based upon the consonants in the word. This key assigns the same digit to letters that have similar sounds. For example, the name Smith can be spelled as:

Smith
Smythe

The SOUNDEX() code for both spellings of Smith is S530. If the .DBF file is indexed on the SOUNDEX() code, all similar sounding words would be grouped together.

SOUNDEX() is not foolproof as it is possible to create similar SOUNDEX() codes from very different spellings and words. For example, Cajun and Cigna are pronounced differently, but both have the identical SOUNDEX() code of C250.

SOUNDEX() is useful for applications dealing with names. It should not be used as the sole lookup but should be combined with some sort of window selection. Listing 19.7 illustrates a SOUNDEX() lookup function to get a name from the database.

Listing 19.7 SOUNDEX() lookup

```
#include "inkey.ch"
local mname:=space(25), sxname:=""
use CUSTOMER new
index on soundex(CUSTOMER->Name) to cust1
@ 10,10 say "Name..: " get mname
read
if lastkey() <> K_ESC
   sxname := soundex(mname)
   seek sxname
   do while sxname == soundex(CUSTOMER->Name)
      ? CUSTOMER->Name
      skip +1
   enddo
endif
```

Multiple lookup keys

Sometimes more than one index is maintained on a .DBF file. When this occurs, the user may need to look up a key from an index other than the controlling index. To allow the user to do this, you must switch the controlling index to the one the user needs. This is done with the SET ORDER TO command. The syntax is:

```
SET ORDER TO <nOrder>
```

<nOrder> is the ordinal position of the index within the SET INDEX or USE command when the file was opened. For example:

```
#include "inkey.ch"
use CUSTOMER index By_state, By_sales, By_rep new
@ 10,10 say "Enter state ...: " get m_state
@ 11,10 say "  or Sales Rep.: " get m_rep
read
if lastkey() <> K.ESC
   if !empty(m_state)
      set order to 1
      seek m_state
   elseif !empty(m_rep)
      set order to 3
      seek m_rep
   else
      ? "Must enter a rep or a state code"
   endif
endif
```

The controlling index order can be determined by using the INDEXORD() function. Its syntax is:

```
<number> := INDEXORD()
```

The function will return the ordinal number of the controlling index. If zero is returned, the records are not in any logical order controlled by an index, but rather are in record number order.

Optimizing multiple searches

If the user is searching for records meeting two conditions that are both in index expressions, it is necessary to try to determine which expression should be sought first. Usually, whichever condition is likely to have the fewest number of records should be sought first.

For example, assume we had a database of all registered voters in the nation. The database is indexed by sex and state. The user needs to find all women in North Dakota. This problem could be solved by SEEK “F”, reading those records and displaying the ones in North Dakota. Since the population of the United States is more than fifty percent women, this search would read through at least half of the database. If the search were reversed, and the state was looked for first, the number of records would be considerably smaller (since less than fifty percent of the population lives in North Dakota.)

Whenever the user needs to look up information which is available in two index files try to make sure the index with the smaller number of entries is searched first.

Keyword files

A keyword is a word extracted from a larger string that can be used to refer to that string. For example, if you were in a library looking for books on Clipper, you would be interested in any of the titles in Figure 19.4.

Figure 19.4 Sample Clipper Books

Using Clipper
Clipper 5: A Developer's Guide
Developing Applications with Clipper

Unfortunately, there is no easy way to represent the above titles in an index file and still be able to access them by the keyword ***Clipper***. To solve this type of lookup problem, we need to create a keyword file.

A keyword file is a separate .DBF file which contains the appropriate keyword and a pointer to the record in the primary .DBF file. The keyword file is indexed by keywords and it is searched against rather than the main .DBF.

Let's illustrate an example that might allow a video store to look up movies based upon any word from the title. Figure 19.5 contains the database structures we will use.

Figure 19.5 Keyword file .DBF structures

```
File:   VIDEOS.DBF
Index:  Upper(id_code)

 1. TITLE          Char          40   Video title
 2. RATING         Char           4   Movie rating G,PG,PG13,R,X
 3. PRICE          Numeric      6,2   Purchase price
 4. ID_CODE        Char           6   Unique code for each video

File: KEYWORDS.DBF
Index:  Upper(keyword)

 1. KEYWORD        Char          12   Single keyword
 2. ID_CODE        Char           6   Pointer into VIDEOS.DBF
```

As each video is added to the VIDEOS.DBF, an entry is made in the keyword file for each word within the title. If the movie FATAL VISION were added to the VIDEOS.DBF and assigned the code of FATVIS, the two entries added to the keyword file would be:

Keyword	*Id_code*
FATAL	FATVIS
VISION	FATVIS

When the user requests that "something VISION" movie, the system would seek "VISION" in the keyword database, and find the ID_CODE of FATVIS. This ID_CODE would be looked up in the primary database to display the movie title.

Listing 19.8 contains two functions useful for keyword files. The Keyw_Build() function writes entries into the keyword file for a title and code. Its syntax is:

```
Keyw_Build(cTitle,cId_code)
```

<cTitle> is the movie title to be added. ***<cId_code>*** is the unique code assigned to this movie.

The Keyw_Find() function returns an array of titles based on the keyword. Its syntax is:

```
<aTitles> := Keyw_Find(cWord)
```

<cWord> is the keyword that the database should be searched for.

Listing 19.8 Keyword functions

```
function Keyw_Build(cTitle,cId_code)
local skip_words:={"A","THE","OF","AND"}
local words:= Parse(upper(cTitle)," ")  // Parse() is from
local jj                                // Chapter 20.
use KEYWORD new index KEYW
for jj := 1 to len(words)
   if ascan(skip_words,words[jj]) == 0
      append blank
      replace keyword with words[jj],id_code with cId_code
   endif
next
return nil

function Keyw_Find(cWord)
local arr_ :={}
use VIDEOS new index VIDS
use KEYWORD new index KEYW
```

```
seek upper(cWord)
do while !eof() .and. upper(cWord) == KEYWORD->Keyword
   select VIDEOS
   seek KEYWORD->Id-code
   if found()
      aadd(arr_, VIDEOS->Title+' '+VIDEOS->Id-code)
   endif
   select KEYWORD
   skip +1
enddo
return arr_
```

Querying the file

Clipper provides commands that allow you to collect information about all records or a subset of records in a .DBF file. These commands process records in the database and perform the indicated operation, either counting, summing fields, or computing their average values. For more complex querying, the DBEVAL() function allows a block of code to be executed for each record in the database.

COUNT

The COUNT command is used to determine the number of records in the database which meet the specified condition. Its syntax is:

```
COUNT TO <nVar> <scope> WHILE <condition> FOR <condition>
```

<nVar> is required and indicates the numeric variable where the results of the count operation should be stored. If <nVar> has not been previously declared, COUNT will create it as a **private** variable. Its scope will be the current procedure and all procedures called by the current one.

<scope> is optional and indicates which records should be counted. It may be any one of the values in Table 19.1. If the <scope> is not specified, the default is ALL records.

The ***WHILE <condition>*** is optional and indicates a logical condition that each record is compared against. If the condition is true, the count operation continues. Once a false condition occurs, COUNT immediately stops.

The ***FOR <condition>*** is also optional and indicates a logical condition that each record is compared against. If the condition is true, then that record is included in the count. If false, the record is not included. Regardless of whether or not the record is included, COUNT continues to count records until the end of the scope or a failed WHILE condition.

Listing 19.9 illustrates a few examples of the COUNT command.

Listing 19.9 COUNT examples

```
use CUSTOMER new
count to how_many                        // All customers
count to ca_cust for State == "CA"  // California customers
index on State to cust1
seek "ID"
count to id_cust rest while State == "ID" // Idaho customers
```

COUNT can also be used to check for imbedded EOF markers. An imbedded EOF marker is a condition which can occur in a .DBF file and cause indexing problems. It usually occurs when the system crashes without properly closing all open files. The function in Listing 19.10 illustrates the use of COUNT to check for EOFs.

Listing 19.10 Imbedded EOF check

```
function CheckEof(DBF_file)
local actual_ct:=0                  // Compare actual count with
use (DBF_file) new                  // count stored in .DBF header.
count to actual_ct                  // If they are not the same,
return (actual_ct<>Lastrec())       // probably an imbedded EOF.
```

If an imbedded EOF exists, it can be corrected by following the procedures outlined in Figure 19.6.

Figure 19.6 Removing imbedded EOF markers

1. Copy all records up to the EOF marker into a new file.

```
select BAD_FILE
COPY ALL TO NEW FILE1
```

2. Skip past the EOF marker.

```
skip +2
```

3. Copy the remaining records to a new file.

```
COPY REST TO NEWFILE2
```

4. Combine the two files.

```
select NEWFILE1
append from NEWFILE2
```

5. Rename the old file and replace it with the new file.

```
close databases
rename BAD_FILE.DBF to BAD_FILE.OLD
rename NEWFILE1.DBF to BAD_FILE.DBF
```

SUM

The SUM command is used to total up certain fields from the database. Its syntax is:

```
SUM <exp> TO <vars> <scope> WHILE <condition> FOR <condition>
```

<exp> is a single field expression or a list of expressions which will be summed. Each expression must have a corresponding variable in the ***<vars>*** list. If the list of expressions and the variable list do not agree in number, a compilation error will result.

<vars> is a single field variable or a list of variables which will be used to hold the results of the data summed. Each variable will hold the sum of the corresponding expression in the ***<exp>*** list.

<scope> is optional and indicates which records should be summed. It may be any one of the values in Table 19.1. If the <scope> is not specified, the default is ALL records.

The ***WHILE <condition>*** is optional and indicates a logical condition that each record is compared against. If the condition is true, the SUM operation sums this record to a variable and continues. Once a false condition occurs, SUM immediately stops.

The ***FOR <condition>*** is also optional and indicates a logical condition that each record is compared against. If the condition is true, then that record is included in the sum. If false, the record is not included. Regardless of whether or not the record is included, SUM continues to process records until the end of the scope or a failed WHILE condition.

Listing 19.11 illustrates some examples of the SUM command.

Listing 19.11 SUM examples

```
use CUSTOMER new
// Balance due for all customers
sum Balance to total_due
// Balance due for CALIFORNIA customers only
sum Balance to calif_due for State == "CA"
// Using WHILE and indexes to speed up the sum command
index on State to cust2
seek "PA"
sum rest Balance to penn_due while State == "PA"
```

AVERAGE

The AVERAGE command is used to calculate averages for numeric fields in a database. Its syntax is:

```
AVERAGE <exp> TO <vars> <scope> WHILE <condition> FOR <condition>
```

<exp> is a single field expression or a list of expressions which will be averaged. Each expression must have a corresponding variable in the ***<vars>*** list. If the expression list and variable list do not agree in count, a compilation error will be generated.

<vars> is a single field variable or a list of variables which will be used to hold the results of the data averaged. Each variable will hold the average computed by the corresponding expression in the ***<exp>*** list.

<scope> is optional and indicates which records should be averaged. It may be any one of the values in Table 19.1. If the <scope> is not specified, the default is ALL records.

The ***WHILE <condition>*** is optional and indicates a logical condition that each record is compared against. If the condition is true, the AVERAGE operation continues. Once a false condition occurs, AVERAGE immediately stops.

The ***FOR <condition>*** is also optional and indicates a logical condition that each record is compared against. If the condition is true, then that record is included in the average. If false, the record is not included. Regardless of whether or not the record is included, AVERAGE continues to average records until the end of the scope or a failed WHILE condition.

Listing 19.12 illustrates some examples of the AVERAGE command.

Listing 19.12 AVERAGE examples

```
use CUSTOMER new
// Average balance due for all customers
average balance to total_due
use INVOICES new
// Average past due balances
average inv_total to past_due for ;
      date() - inv_date > 45
// Average PA salesman over $1,000 in sales
use SALESMEN new
index on State to sale2
seek "PA"
average rest total_sales to pa_avg ;
      for total_sales > 1000 while State == "PA"
```

Efficient processing

One of the dangers of the query operations is that they can cause considerable disk activity. The functions should be used with care to prevent extra disk work. For example, consider the request to count and total all sales within the state of California. Listing 19.13 illustrates one way to approach this problem.

Listing 19.13 COUNT and TOTAL Sales

```
select SALESREP new
count to Sales_ct all for State == "CA"
sum Ytd_sales to tot_sales for State == "CA"
```

This approach will work, but also requires two passes through the database. The same effect can be accomplished as shown in Listing 19.14. Although this listing contains more lines of code, it is actually more efficient since the database is processed only once.

Listing 19.14 COUNT and TOTAL sales

```
local x, sales_ct:=0, tot_sales:=0
select SALESREP new
x:=lastrec()
for k:=1 to x
   if State == "CA"
    sales_ct++
    tot_sales += Ytd_sales
    skip
   endif
next
```

DBEVAL()

The DBEVAL() function can be used to process records in a .DBF file and apply a set of code to each record. The COUNT, SUM, and AVERAGE commands are all converted to DBEVAL() function calls by the preprocessor.

The syntax for DBEVAL() is:

```
DBEVAL(<block>,<bFor>,<bWhile>,<nNext>,<nRecord>,<lRest>)
```

<block> is the code block to evaluate for each database record. This block of code is executed for each record in the .DBF file which meets the conditions set up by the other code blocks.

<bFor> and ***<bWhile>*** are code blocks that correspond directly to the FOR and WHILE clauses of the SUM, COUNT, and AVERAGE commands. Basically, if you use either or both of these clauses, DBEVAL() will process records until the code blocks return false (.f.).

<nNext> and ***<nRecord>*** are both numerics. The value of <nNext> specifies how many records to process from the current record. The <nRecord> specifies which record number to process.

<lRest> is a logical value that determines whether the DBEVAL() scope will be from the current record to the end-of-file, or all records. If you pass true (.t.), DBEVAL() will assume that you prefer the former (i.e., start from current record). If you pass false or ignore this parameter, DBEVAL() will process all records.

DBEVAL() is discussed further in Chapter 8, "Code Blocks."

Creating subsets

Database files are collections of related records. For example, a customer .DBF file probably contains all clients for a particular company. Frequently, however, operations need to be performed on a group of records, not the entire file. This group can be referred to as a subset file.

A subset file is a smaller view of a larger file. It is used to allow a set of operations to be performed on only certain records from the file or to view only the selected records.

There are several approaches in Clipper which can be used to create subset files.

SET FILTER

The SET FILTER command allows the programmer to create a logical subset of a work area. This logical view is not a separate file, but a condition applied to a work area. The condition hides records which do not apply. This causes the database to appear to contain only a small subset of records.

The command syntax is:

```
SET FILTER TO <condition>
```

<condition> is an expression which can be evaluated to a logical result (true or false). Each record is tested against the expression. If the result is true, the record is displayed and appears to be part of the file. If the result is false, Clipper ignores the record. For example:

```
set filter to CUSTOMER->State == "PA"
```

This command creates a logical view where only the customers in the state of Pennsylvania are displayed.

The advantage of the SET FILTER command is its ease of use. The SET FILTER command automatically forces other Clipper commands to view the database as a subset of records. In addition, updates can be applied directly to the records in the database. If the subset were a separate file, any updates performed on the subset would have to be applied back to the original files.

The disadvantage is speed. SET FILTER can be very SLOW!! Most programmers avoid using SET FILTER for this very reason. When the filter creates a subset of records, all records in the dababase must be evaluated to decide whether or not to display the record. Since SET FILTER does not intelligently work with index files, a lot of extra disk reads may be necessary while trying to find a record which meets the filter condition. If the filter creates a subset of 100 records out of 5,000, then 4,900 records will be processed and ignored in order to display the 100 records.

While completely avoiding the command is a bit extreme, SET FILTER should be used with utmost caution. To the end-user, the database appears very small. This makes it difficult to explain why it takes so long to display only 50 records or so.

A filter can be used efficiently if the conditions are right. If the database contains many records which meet the filter criteria, and only a few records will be hidden, SET FILTER can be very powerful. For example, consider a voter registration database which contains a single character code for party affiliation. The possible codes are listed in Figure 19.7.

Figure 19.7 Party codes

D -	Democrat
I -	Independent
R -	Republican

The command:

```
set filter to party $ "RD"
```

could be used to display only those voters affiliated with the Democratic or Republican parties. This filter would operate fairly efficiently (assuming most voters are Democrats or Republicans.)

The filter:

```
set filter to party == "I"
```

would probably not be a fast way to display only the independent voters.

Once a filter is created, it stays applied to the current work area until the work area is closed or a new filter is applied. The SET FILTER command can be used without any parameters to remove a filter condition from a work area. The syntax to remove a filter is:

```
SET FILTER TO
```

Saving filter queries

Once a filter is created within a work area, it can be saved as part of the work area's status information. The .DBFILTER() function returns the filter condition as a character string. Its syntax is:

```
<cFilter_expression> := .DBFILTER()
```

The ***<cFilter_expression>*** string contains the expression that was applied when the SET FILTER command was executed. This variable can be treated just like any other variable. It may be saved to a memory variable file as well. The program in Listing 19.15 illustrates a work area being closed to allow a DOS program to be run and then being reopened. Once it is reopened, the filter condition is reapplied to the database.

Listing 19.15 Saving filters

```
use CUSTOMER new
set filter to CUSTOMER->State == "OR"
*
* Some lines of code
*
cust_filt := .DBFilter()
close database
run RR          //Run external report writer
use CUSTOMER new
set filter to &cust_filt.
```

Create separate subset files

If the filter command will result in unacceptable performance, you can also create the subset file as a separate .DBF file. This will require copying the structure and the appropriate records from the primary file into the subset. Clipper provides several ways to do this.

COPY TO

The COPY TO command is used to create a new file which contains a selected set of records from the current work area. Its syntax is:

```
COPY FIELDS <flist> TO <new_DBF> <scope> ;
        WHILE <condition> FOR <condition>
```

FIELDS <flist> specifies a list of fields from the current work area to be copied into the new file. The field names are separated by commas. If the FIELDS keyword is not specified, all fields are copied into the new file.

The ***WHILE <condition>*** is used to narrow down the records which might be returned by the COPY command. The condition specified after the WHILE keyword indicates how long the search should continue. As soon as a record fails to meet the WHILE condition, the COPY command stops.

The ***FOR <condition>*** is also optional and indicates a logical condition that each record is compared against. If the condition is true, then that record is copied to the new file. If false, the record is not copied. Regardless of whether or not the record is copied, COPY continues to transfer records until the end of the scope or until a failed WHILE condition.

Listing 19.16 illustrates some examples of the COPY TO command.

Listing 19.16 COPY TO examples

```
use CUSTOMER new
// copy all fields from CUSTOMER to drive A:
copy to a:CUSTOMER
// copy name and phone number to a .DBF file
copy fields Name,Phone to CUSTPHON

// Copy Pennsylvania customers to a separate .DBF
index on State to cust1
seek "PA"
copy rest to pa_custs while State == "PA"
```

SORT

The SORT command can also be used to create a subset database, with the added benefit of creating the subset in sorted order. The SORT command is described at the beginning of this chapter. Regardless of how the subset file is created, it is now available for processing. The biggest drawback with a separate file is that updates applied to the file must be made back to the original file as well.

To write back the information to the original file, you must determine where the original record numbers were. Listing 19.17 illustrates an example of creating a subset database and writing the changes back to the original file.

Listing 19.17 - Subset database example

```
field state, id_code
local jj, flds
use CUSTOMER new
index on State+Id_code to cust
seek "PA"
copy to PA_CUST while State == "PA"       // Create subset
use PA_CUST new                           // Use the subset file
go top
Browse()                   // Allow user to work with subset
flds := fcount()
go top
do while !eof()
   select CUSTOMER
   seek PA_CUST->State+PA_CUST->Id-code
   if eof()
      append blank
   endif

   for jj := 1 to flds
     Fieldput(jj,PA_CUST->(fieldget(jj)) )
   next
   select PA_CUST
   skip +1
enddo
```

Using arrays for subset files

If the number of records to be extracted from the database is small, they can be copied into an array in memory. This array can then be manipulated in a manner similar to a database. Since the array is in memory, the performance is very speedy. In addition, the record numbers can be stored with the array so updates can be made as each array element is processed.

To illustrate the use of arrays for file subsets, let's work through an example. An imaginary company that publishes xBASE compilers has a large file of all its employees. Each employee is assigned to a department. We need to create a program that will allow the manager to update individuals in his or her department. The pertinent fields from the employee file are shown in Figure 19.8.

Figure 19.8 Employee fields

Dept_id	Char	6	ACCT/TESTING/SUPPORT / SALES/PROGRAMS
Full_name	Char	25	Employee's Full Name
Hired	Date	8	Date of hire
Salary	Real	12,2	Yearly Salary

Figure 19.19 shows the screen the manager needs to see.

Figure 19.19 Update screen

Dept:		
Employee Name	Hire Date	Salary

When the department is entered, the screen should be filled with the names, hire dates, and salaries of all employees within that department. Assuming that the employee file is indexed on DEPT_ID+FULL_NAME, the code fragment in Listing 19.19 can be used to create an array of employees.

Listing 19.19 Employee Array

```
#include "inkey.ch"
local mdept:=space(6)
private employees:={}      //Must be private to be visible in
                           //ACHOICE() udf
use EMPLOYEE new index DEPT_ORD
@ 2,6 get mdept pict "!!!!!!"
setcursor(1)
read
if lastkey() <> K_ESC
   seek mdept
   do while !eof()  .and. EMPLOYEE->Dept_id == mdept
      aadd(employees,full_name+"  "+dtoc(hired)+" "+;
      str(salary,12,2)+"  "+str(recno(),6) )
      skip +1
   enddo
endif
```

Notice that during the array add command, the record number from the employee file is included in the array. We have just created a memory structure to hold the employee information that needs to be displayed upon the screen. Now that the structure has been created, we can use ACHOICE() to display the information. Each element in the array is 56 characters long. However, we do not want to show the last 6 characters, i.e. the record number. Therefore, we call ACHOICE() with a 50 character wide window. ACHOICE() will automatically not display the last six characters, although they are still accessible in the array.

Listing 19.20 illustrates the use of ACHOICE() to perform simple data update.

Listing 19.20 ACHOICE() data update

```
#include "inkey.ch"
// The variables keepgoing, elt, and pos will be updated
// by the A_UDF called from ACHOICE. As a result, they
// must be declared as PRIVATE rather than LOCAL.

local x, mname, mdate, mamt, mrow,mrec
private keepgoing := .t., elt:=1,pos:=0, tags[len(employees)]
afill(tags,.t.)
do while keepgoing
   x :=achoice(6,15,22,65,employees,tags,"A_UDF",elt,pos)
   if x > 0
      mname := substr(employees[x],1,25)
      mdate := ctod(substr(employees[x],28,8))
      mamt  := val(substr(employees[x],37,12))
      mrec  := val(right(employees[x],6))
      mrow  := row()
      @ mrow,15 get mname
      @ mrow,42 get mdate
      @ mrow,51 get mamt
      read
      if lastkey() <> K_ESC
         select EMPLOYEE
         goto mrec
         replace EMPLOYEE->full_name with mname,;
                 EMPLOYEE->hired with mdate,;
                 EMPLOYEE->salary with mamt
         employees[x] :=mname+"  "+dtoc(mdate)+" "+;
                        str(mamt,12,2)+"  "+str(mrec,6)
      endif
   endif
enddo
```

Since the data is stored in an array, things can be done that otherwise might be too slow. The A_udf() function in Listing 19.21 illustrates a couple of simple features.

- A search feature for employee name, hire date, or salary range. If the manager wants to see all the employees who were hired during a certain year and are paid below a minimum salary, the program should be able to display only the records meeting the criteria.

- A feature to sort the array on any one of the three fields. With the code block parameter on ASORT(), positional sorts of an array are extremely easy to code.

Listing 19.21 ACHOICE() user-defined function

```
#include "ACHOICE.CH"
#include "INKEY.CH"

function A_udf(mode,element,position)
local lk:=lastkey()
memvar tags

// Private variables ELT,POS, and KEEPGOING are
// passed from the program containing the ACHOICE
// function call.

private elt, pos, keepgoing          //squelch compiler warnings
elt := element                       // update element
pos := position                      // and position
if mode == AC_EXCEPT                 // Keystroke exception
   do case
   case lk == K_ESC
      keepgoing := .f.               // Tell LOOP to exit
      return AC_ABORT
   case lk == K_ENTER .or. lk == 32  // ENTER or space
      return AC_SELECT
   case chr(lk) $ "Ss"                 // Sort array
      Sort_it()
      return AC_ABORT
   case chr(lk) $ "Ff"                 // Find an item
      Find_it()
      return AC_ABORT
   endcase
endif
return AC_CONT
```

Sorting the array

If the user presses “S” to sort, the first thing to do is determine which field to sort on. Once we determine the field, we need to sort the array, and then return control to A_udf(), which will return a zero and cause the original routine to display the sorted records. Listing 19.22 illustrates the Sort_it() function.

Listing 19.22 Sort_it()

```
function Sort_it
local which := 1
@ 6,15 clear to 22,65
@ 6,15 prompt "Name"
@ 6,43 prompt "Date"
@ 6,52 prompt "Salary"
menu to which
do case
case which == 1
   asort(employees)
case which == 2
   asort(employees,,,{|x,y|substr(x,28,8) < substr(y,28,8)} )
case which == 3
   asort(employees,,,{|x,y|substr(x,37,12) < substr(y,37,12)} )
endcase
return nil
```

Finding a further subset

The FIND command we've described above actually reduces the subset of editable employees even further. To handle this step, we first need to determine the criteria to select on. Once we determine the criteria, we fill the **tags** array with .f. values, making every employee non-selectable. We then read through the array, and if the employee matches the criteria we will set the corresponding **tags** array element to .t. When the array is redisplayed, only the items we tagged that met the criteria will be editable. The code for Find_it() appears in Listing 19.23.

Listing 19.23 Find_it()

```
#include "inkey.ch"
function find_it
local mname:=space(25), myear:=year(date())
local mop_code :="<", mamt:=0.0, jj, tag_me, find_one:=.f.
@ 6,15 clear to 22,65
@ 6,15 get mname
@ 6,48 get myear     pict "99"
@ 6,51 get mop_code  pict "!"  ;
       valid mop_code $"<>= " .or. lastkey() == K_UP
@ 6,52 get mamt      pict "999999.99"
read
afill(tags,.f.)
for jj:=1 to len(tags)
   tag_me := .t.
   if !empty( mname )
     tag_me := (upper(mname) $ upper(substr(employees[jj],1,25)))
   endif
   if !empty( myear )
     tag_me := (year(ctod(substr(employees[jj],28,8))) == myear)
   endif
   if !empty( mamt )
     x := val(substr(employees[jj],37,12))
     do case
     case mop_code == "<"
        tag_me := (x < mamt)
     case mop_code == "="
        tag_me := (x == mamt)
```

```
         case mop_code == ">"
            tag_me := (x > mamt)
         endcase
      endif
      tags[jj] := tag_me
      if tag_me
         find_one :=.t.
      endif
   next
   if !find_one
      afill(tags,.t.)
   endif
   return nil
```

Summary

By the time you finish reading this chapter you should be able to find your data no matter how well it is hidden. You should be familar with various searching techniques and when to employ each one. You should also understand how Clipper allows data querying and be aware of the potential inefficiencies in it. Finally, you should be comfortable with various methods for handling subsets.

CHAPTER 20

Manipulating Character Strings

A string is a collection of characters stored together in memory. Both the characters and their order are maintained by the system. Clipper provides functions and operators which can specifically be used to manipulate string variables. In this chapter, we will discuss how strings are created and handled internally and the functions and operators provided for strings by Clipper. Finally, we will provide some useful user-defined functions to manipulate strings.

String variables

A string may be a single character or a group of characters. The largest string that Clipper supports is 65,519 (64K) bytes long. 64K is actually 65,536 bytes, but Clipper reserves 16 bytes for its own use and terminates the string with a null character. Clipper considers a string and the contents of a memo field to be identical for operating purposes. They are stored differently on disk, but once in memory Clipper treats them the same. This allows all functions which operate on strings to be applied to memo variables and vice versa.

Strings are a very powerful part of Clipper. Many of the commands and functions are designed to expect a character expression rather than a literal option. This allows you to use a string variable or a literal constant. This concept extends deeply into the language and contributes to its richness. A good example can be found in the picture clause of the GET command. The syntax for the command is:

```
GET <var>  PICTURE  <character string>
```

Since the picture clause is a string rather than a hard-coded literal, picture formats can be data driven. A United States zip code picture format might be "99999-9999" while the Canadian zip code format is "A9A 9A9". By creating a string variable called ZIP_FMT, we can have the same GET clause operate differently depending upon the initialization parameters specified by the user.

Creating string variables

A string is created in Clipper using the assignment operator. The assignment data may take several forms: A literal constant, a function which returns a string, a field from a database, or any combination of the three.

Literal constants

A literal constant is a string of characters surrounded by a set of delimiters, for example:

```
string := "CLIPPER is a powerful language"
```

Clipper provides three sets of delimiters. Since the delimiter character could be part of the string itself, providing three sets allows most strings to be represented. The three sets of delimiters are:

[]	Left and right square brackets
""	Two double quotes, one to start and one to end
''	Two single quotes, one to start and one to end

A null string should be represented by a set of delimiters with no characters in between. For example:

```
string1  := []
string2  := ''
```

To represent a string which contains the delimiter characters, you can use an alternate delimiter set. For example, the string

```
Clipper 5: A Developer's Guide
```

could be represented as:

```
[Clipper 5: A Developer's Guide]
```

or

```
"Clipper 5: A Developer's Guide"
```

The sentence:

```
He said, "Clipper is GREAT!!!"
```

could be stored as

```
[He said, "Clipper is GREAT!!!"]
```

By using the three sets of delimiters almost any expression can be represented as a string constant. In the unlikely event that all three delimiters are needed, you can use the CHR() function instead of the literal character. The ASCII values for the delimiter characters are:

"	Double quote	34
'	Single quote	39
[	Left bracket	91
]	Right bracket	93

To represent the string:

```
He said, "Tom's code uses []."
```

in Clipper, we would write:

```
string := [He said,"Tom's code uses ]+chr(91)+chr(93)+[."]
```

Fortunately, strings containing all three delimiters do not occur very often.

Functions which return a string value

A string may also be created from a function that returns a string value. For example:

```
clip_ver := Version()     // Returns CLIPPER version
cur_proc := Procname()     // Returns current procedure name
dos_ver  := Os()          // Returns operating system name
```

(See Table 20.6 for a complete list of Clipper functions which return string values.)

A field from the database

A string may also be used to hold the contents of a character field or a memo field from a database. Assume we had the following fields in a database called CUSTOMER:

```
1. Comp_name          Char  25
2. Notes              Memo  10
```

The following assignments would transfer the company name and notes from the current record into string memory variables.

```
mcompany := CUSTOMER->Comp_name
mnotes   := CUSTOMER->Notes
```

A string may also be created by combining constants, functions, and database fields. For example, the following code fragment would produce a valid Clipper string:

```
string := "Clipper "+Version()
```

Once a string is created it can be used and manipulated by Clipper. Many functions and operators are available specifically for dealing with string variables.

String operators

Clipper provides operators to concatenate strings and to compare the sort order of strings. These operators are described in the following section.

Concatenation

Concatenation is the combining of two strings to form a new string. Clipper provides two operators for concatenating strings. These operators are the plus (+) and the minus sign (-) and are different only in the way they handle trailing spaces. The general syntax is shown below:

```
string  :=   <string1> <operator> <string2>
```

If the operator is the plus sign, the contents of **string2** are added to the end of **string1** and the result would be placed into **string**. If the operator is the minus sign, **string1** has all trailing spaces removed and appended onto the end of **string2**. Then the two strings are combined. Let's look at some examples:

```
? "Hello " + "World"
```

would display:

```
Hello World
```

while:

```
? "Hello " - "World"
```

would display:

```
HelloWorld
```

The minus operator is handy when you wish to combine first and last names. For example, assume you had a database containing the following fields:

```
1.  First       Char 12
2.  Last        Char 15
```

JOE is stored in the field FIRST and BOOTH stored in the field LAST. This is how the two concatenation operators would handle these fields

```
? FIRST + LAST     //would produce
 J O E                   B O O T H
 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 < 12 characters       >< 15 characters              >
```

while:

```
? FIRST - LAST    //would produce
 J O E B O O T H
 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 < 27 characters                                     >
```

Both strings would still be 27 characters long, the difference being the space characters are positioned at the end of the string.

Clipper 5 introduced a new syntax for these operators which may be used interchangeably with the plus and minus operators. The plus sign can be written as += and the minus sign as -= . The following examples illustrate this shorthand notation.

`FIRST += LAST`	is the same as	`FIRST = FIRST + LAST`
`FIRST -= LAST`	is the same as	`FIRST = FIRST - LAST`

C programmers will immediately recognize this notation, as it is available in most C dialects.

Comparison

String variables can also be compared in a variety of ways using Clipper operators. To understand how strings are compared by Clipper, we need to first look at how a string is stored internally.

Internal string format

Strings are stored in memory (and on disk) using the ASCII character chart (see Appendix A). The ASCII chart contains a complete numerical ranking for all characters which can be represented in a string. Since a computer processes numerical information, a string can be viewed as a list of ASCII numbers. For example, the string "CLIPPER 5" and its ASCII equivalent are shown in Figure 20.1.

Figure 20.1 ASCII equivalent of string "Clipper 5"

```
C     L     I     P     P     E     R           5
67    76    73    80    80    69    82    32    53
___   ___   ___   ___   ___   ___   ___   ___   ___
```

To perform a comparison, the computer compares the numerical ranking of each character within the string. If the computer is asked if "A" is less than "B", i.e.:

```
?   "A" < "B"
```

The computer would determine that the letter **A** is the number 65 on the ASCII chart and the letter **B** is the number 66. Since 65 is less than 66, the comparison would be true, **A** is less than **B**.

Case considerations

Using the ASCII chart allows the computer to compare string variables. It also introduces some possible case confusions. For example, the ASCII code for upper case letter **B** is 66, while the code for lower case letter **A** is 97. Using these numbers, the comparison:

```
? "a" < "B"
```

would return false, since this is how they are ranked on the ASCII chart. Clipper provides case conversion functions to make comparisons of this sort more meaningful. For example, you could use:

```
? lower("a")  <  lower("B")
```

which would convert each string to lower case before comparing them. In the above case, the comparison would now produce true.

Are two strings equal?

Two operators are used to compare strings for equality. These operators are different in the way they handle strings of varying lengths. The operators are the equal sign (=) and the exactly equal sign (==). In addition, the SET EXACT command affects the way the equal sign performs.

Equal sign. The equal sign is affected by the SET EXACT command. The syntax of the SET EXACT command is:

```
SET EXACT < OFF | ON | logical >
```

The default value is OFF. SET EXACT is also considered a compatibility command that may not be supported in future releases of Clipper.

When SET EXACT is OFF, the rules in Table 20.1 are used to compare two strings:

Table 20.1 SET EXACT OFF

1. If the string on the right is null, return true.

2. If the string on the right is longer than the string on the left, return false.

3. For the length of the right hand string, compare each character with the corresponding character position in the string on the left. If each character is identical, return true. If not, then return false. It does not matter if the string on the left is larger than the string on the right.

If set exact is ON, the rules in Table 20.2 are used for comparison.

Table 20.2 SET EXACT ON

1. Remove trailing spaces from both strings.

2. If the strings are not the same length, return false.

3. Compare each character position in the two strings. If any characters are different return false. If all characters are the same, return true.

Let's review some examples:

```
SET EXACT OFF

? "ABC"    = "ABCDEF"      // FALSE - Table 20.1, rule 2
? "ABCDE"  = "ABC"         // TRUE  - Table 20.1, rule 3
? "ABC"    = ""            // TRUE  - Table 20.1, rule 1
? ""       = "ABC"         // FALSE - Table 20.1, rule 2
```

```
SET EXACT ON

? "ABC"    = "ABCDEF"       // FALSE  - Table 20.2, rule 2
? "ABCDE"  = "ABC"          // FALSE  - Table 20.2, rule 2
? "ABC"    = ""             // FALSE  - Table 20.2, rule 2
? "ABC"    = "ABC"          // TRUE   - Table 20.2, rule 3
```

This handling of string comparisons has some subtle nuances that can cause some problems. For example, try to determine what the following comparison will yield:

```
set exact off
mvar := 50
? str(mvar,3) = " "
```

At a quick glance it appears that this equation will return false, however, the opposite will happen. The **STR(mvar,3)** function will produce a string of three characters, the first of which is a space. Since the comparison checks only one character (the length of the string on the right), both strings contain a space in the first position and hence the expression would return true.

The use of SET EXACT can make the code very confusing to read. It is best to leave SET EXACT to OFF at all times. This is the default setting, so it is not necessary to explicitly code it in your program. If you need to perform a comparison of two strings that must be the same length, use the syntax:

```
trim(<cString1>) == trim(<cString2>)
```

The exactly equal operator (==) is explained in the next section.

Exactly equal sign. The exactly equal sign is not affected by the setting of SET EXACT. It is used for a more exact comparison since the strings being compared must match exactly in both content and length. The rules in Table 20.3 describe how the double equal operator works.

Table 20.3 Rules of the == operator

```
1. If the lengths of the strings are not the same, then return false.

2. For the length of the strings, compare each character position
   in the right string with the corresponding character in the
   string on the left. If each character is identical, return true.
   If not, then return false.
```

Notice that trailing spaces are not removed before the comparison is performed. The strings must be identical including trailing spaces for the exactly equal operator to return TRUE.

Are strings not equal?

There are three operators which may be used to denote not equal comparison. All three operators are the same and can be used interchangeably. The not equal operators are affected by the setting of SET EXACT as well. The three operators and some examples are shown below:

```
!=   All three of these operators have the same
<>   meaning:   Are two strings not equal?
#
```

Examples:

```
if PAYROLL->Department <> "SALES"

log_out := password != "XYZ123"
```

The not equal operator is the inverse of the equal operator. It is affected by the value of SET EXACT. The rules in Tables 20.1 and 20.2 are applied in inverse fashion.

Is one string contained within another?
The **contains** ($)operator tests to see if one string is contained within a larger string. The smaller string is generally referred to as a substring. The syntax for **contains** is:

```
<substring> $ <string>
```

True will be returned if the substring can be found at any position in the string. If not, false will be returned. The search is case sensitive. For example:

```
?  "A"    $ "ABCDEF"   // Returns true
?  "Z"    $ "ABCDEF"   // Returns false
?  "a"    $ "ABCDEF"   // Returns false
```

The **contains** operator can be very handy when validating input to a GET variable. For example:

```
@ 10,10  say  "(P)rinter/(S)creen"
         get  choice pict "!"   valid choice $ "PS"
```

(See Chapter 26 for more details about GET validations.)

Comparing strings sort order
Strings may be compared with one another by comparing their ranking on the ASCII chart. Each character position is compared until one ranking is higher or lower than the other. Once a difference is found, the operator returns a true or false value depending on the difference. Appendix A contains an ASCII chart.

Greater Than (>). The right arrow bracket is used to designate a greater than comparison. For example,

```
? "A" > "Z"        // FALSE - A=65 , Z=90
? "JOHN" > "JOE"   // TRUE  J and O are the same, but the letter
                   // H is higher in the chart than E.
```

Greater-Than-or-Equal-to (>=). The right arrow bracket can be followed immediately by an equal sign to designate a greater-than-or-equal-to comparison.

Less-than (<). The left arrow bracket is used to designate a less-than comparison. For example,

```
? "Z" < "A"         // FALSE -  Z=90 , A=65
? "JOE" < "JOHN"    // TRUE  J and O are the same, but the
                    // letter E is lower on the chart than H.
```

Less-Than-or-Equal-to (<=). The left arrow bracket can be followed immediately by an equal sign to designate a less-than-or-equal-to comparison.

Operator summary

Clipper provides a number of different operations that can be used on strings. These operations are summarized in Table 20.4.

Table 20.4 Operator summary

+	Concatenate two strings.
-	Concatenate and move spaces to end.
=	Are strings equal?
==	Are strings equal and the same length?
#,!=,<>	Are strings not equal?
<	Is one string less than another?
<=	Is string less than or equal to another?
>	Is one string greater than another?
>=	Is string greater than or equal to another?

String functions

In addition to the operators provided, a large number of functions exist which can be used to manipulate strings. These functions can be grouped into five general categories:

TESTING	Test string size, case, contents, etc.
SPACES	Add or remove spaces from string.
EDIT	Change the contents of string characters.
SUBSTRINGS	Create and manipulate substrings.
OTHER	Clipper functions that return a string value.

Testing — test string size, etc.

The testing functions are available to test length and content of string variables.

LEN()

LEN() Determines the number of characters within a string. The size of a string may be determined using the LEN function. Its syntax is:

```
nSize := LEN( <cstring> )
```

nSize will contain an integer which is equal to the total number of characters, including all spaces (leading, trailing, and imbedded). For example:

```
? len("Clipper 5: A Developer's Guide")     // returns 30
? len("")                                   // returns  0
```

The LEN() function can also be used to test the size of character or memo fields from a database. In the case of character fields, LEN() will return the size defined in the database, regardless of the field's contents. In the case of a memo field, LEN() will return the number of characters, including spaces, in the memo field's contents.

EMPTY() — is a string empty?

EMPTY() is a useful function for testing variables of all types. The syntax for EMPTY() is:

```
lEmpty := EMPTY( <exp> )
```

When used with string variables, EMPTY() returns true if the variable contains spaces or is a null string. For example:

```
mpath := getenv("PATH")
if empty( mpath )
   ? "No path is specified..."
endif
```

EMPTY() can also be used to validate a GET that is required. For example, assume the account number must be entered in order for the system to continue:

```
macct := space(8)
@ 10,10 say "ACCOUNT: " get macct valid !empty(macct)
```

The user would be required to answer the account prompt in order to continue. Of course, since the user has no idea why he cannot leave the field, he might also punch the video screen.

ISALPHA() — is the first character alphabetic?

ISALPHA() will check to see if the first character of a string is an alphabetic character. It is the same as the following code:

```
? upper(left(string,1)) $ "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

The syntax for ISALPHA() is:

```
lCheck := ISALPHA( <string> )
```

ISALPHA() could be used to check that the first character of a user-entered file name is alphabetic. For example:

```
@ 10,10 get mfile  valid isalpha(mfile)
```

ISDIGIT() — is the first character a digit?

ISDIGIT() will check to see if the first character of a string is a digit character, between 0 and 9. It is the same as the following code:

```
? left(string,1) $ "0123456789"
```

The syntax for ISDIGIT() is:

```
lCheck := ISDIGIT( <string> )
```

ISDIGIT() could be used to check that the first character of a social security number is numeric. For example:

```
@ 10,10 get mssn  valid isdigit(mssn)
```

This example would not check every character, only the first one. A more comprehensive social security check routine might be coded as illustrated in Listing 20.1.

Listing 20.1 Checkssn()

```
function Checkssn(soc_sec)
local k
for k:=1 to 11
   if .not. (k == 4 .or. k == 7)
      if .not. isdigit( substr(soc_sec,k,1) )
         return .F.
      endif
   endif
next
return .T.
```

This function checks to see that each character in the social security number (except for the dashes) is a valid digit.

ISLOWER() — is the first character a lowercase letter?

ISLOWER() will check to see if the first character of a string is a lowercase alphabetic character. It is the same as the following code:

```
? left(string,1) $ "abcdefghijklmnopqrstuvwxyz"
```

The syntax for ISLOWER() is:

```
lCheck := ISLOWER( <string> )
```

A good example of ISLOWER() is in the Formalize() user-defined function in Listing 20.2.

Listing 20.2 Formalize()

```
function Formalize(sname)
local jj, one_char,upcase:=.t.
for jj:=1 to len(sname)
   one_char := substr(sname,jj,1)
   if upcase .and. islower(one_char)
      sname := stuff(sname, jj, 1, upper(one_char))
      upcase :=.f.
   endif
   upcase := ( one_char == " " )
next
return sname
```

This function takes a string and converts the first letter of each word to uppercase. For example,

```
Formalize("clipper 5: a developer's guide")
```

would return

```
Clipper 5: A Developer's Guide
```

This function can be used to ensure that the data written to a file is consistent, regardless of how the user entered it.

ISUPPER() — is the first character an uppercase letter?

ISUPPER() will check to see if the first character of a string is an uppercase alphabetic character. It is the same as the following code:

```
? left(string,1) $ "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

The syntax for ISUPPER() is:

```
lCheck := ISUPPER( <string> )
```

SPACES — add/remove spaces from a string

Character strings are often used for report and screen headers. Clipper provides a number of functions to manipulate the space characters within a string.

ALLTRIM() — remove leading and trailing spaces

The ALLTRIM() function removes any leading and trailing spaces from a string variable. Its syntax is:

```
<string>  := ALLTRIM( <string> )
```

For example:

```
cstring := "      CLIPPER      "

? len( cString )              // Returns 19
? len(alltrim(cString))       // Returns 7

cString := alltrim(cString)
? cString                      // Displays "CLIPPER"
```

ALLTRIM() is not an actual Clipper function, but rather a pre-processor #translate directive. (See Chapter 7 for more information on the preprocessor). The ALLTRIM() function will be converted to LTRIM(TRIM(<string>)).

LTRIM() — remove leading spaces from a string

The LTRIM() function removes any leading spaces from a string variable. Its syntax is:

```
<string>  := LTRIM( <string> )
```

For example:

```
cstring := "      CLIPPER      "

? len( cString )               // Returns 19
? len(ltrim(cString))          // Returns 13

cString := ltrim(cString)
? cString                      // Displays "CLIPPER      "
```

LTRIM() is useful when displaying STR() values. For example:

```
amount := 1000.00
? str(amount,10,2)           // Displays "   1000.00"
? ltrim(str(amount,10,2))    // Displays "1000.00"
```

RTRIM() — remove trailing spaces from a string

The RTRIM() function removes any trailing spaces from a string variable. Its syntax is:

```
<string>  := RTRIM( <string> )
```

For example:

```
cString := "      CLIPPER      "

? len( cString )           // Returns 19
? len(rtrim(cString))      // Returns 13

cString := rtrim(cString)
? cString                  // Displays "      CLIPPER"
```

RTRIM() may also be written as TRIM().

PADx() — Pad a string with spaces or other character

The PADx() functions add spaces or another pad character to various spots within a string variable. There are three PAD functions: PADC(),PADL(), and PADR(). The syntax of each is:

```
<string>  := PADX( <string>,<length> [,<character>] )
```

PADR() is the inverse to RTRIM(), PADL() to LTRIM(), and PADC() to ALLTRIM().

For example:

```
cString := "Bank Reconciliation"

? padc(cString,30)    // Displays "     Bank Reconciliation      "
? padr(cString,30)    // Displays "Bank Reconciliation           "
? padl(cString,30)    // Displays "           Bank Reconciliation"
```

The optional pad **<character>** can also be used if a character other than space is needed. For example,

```
amount := 1500
? padl(ltrim(str(amount,10,2)),50,"*"
```

This will display

```
******************************************1500.00
```

which is useful when the computer is printing checks.

EDIT — change a string's contents

Clipper provides four functions to change the contents of a string variable. Two of these functions are case conversions that affect the entire string. The other two allow individual characters within a string to be inserted, removed, or changed.

LOWER() — convert all characters to lower case

LOWER() will convert all alphabetic characters in a string to lower case. Its syntax is:

```
string := LOWER( <string> )
```

UPPER() — convert all characters to upper case

UPPER() will convert all alphabetic characters in a string to upper case. Its syntax is:

```
string := UPPER( <string> )
```

These two functions are very useful when searching a database. Index keys may be built in either upper or lower case. When the program seeks a key, it converts the key to the appropriate case. This way is does not matter what case the user enters, the key will still be found. For example,

```
select EMPLOYEE
index on upper(Last_name)  to employ1

mname := "Smith"
seek mname
? found()                     // Returns .F.
seek upper(mname)
? found()                     // Returns .T.
```

STRTRAN() — search and replace string characters

This function allows character patterns to be replaced with a string or memo variable. Its syntax is:

```
<cStr> :=STRTRAN(<cStr>,<cSearch>,<cReplace>,<nStart>,<nCount>)
```

The first two parameters are required. **<cStr>** is the string to search and **<cSearch>** is the pattern to search for. The optional **<cReplace>** is the pattern to replace the searched pattern with. If not supplied, it will default to a null (""), which has the effect of removing the search pattern from the string. For example:

```
mpath :=Getenv("PATH")        //  C:\DOS;C:\UTIL;C:\CLIPPER5
mpath :=strtran(mpath,";",",")
? mpath                       // C:\DOS,C:\UTIL,C:\CLIPPER5

mpath := strtran(mpath,"C:\")
? mpath                       // DOS,UTIL,CLIPPER5
```

Both the search pattern and the replace pattern can be any number of characters in length. It is possible to change the string length with STRTRAN().

The last two parameters indicate which occurrence to start the replacement at, and the number of times to perform the replacement. The default starting position is one. The default number of occurrences is all.

STUFF() — insert and remove string characters

This function allows character patterns to be inserted and removed within a string or memo variable. Its syntax is:

```
<cStr> := STUFF(<cStr>,<nStart>,<nDelete>,<cInsert>)
```

The function returns the modified string. It can be used for inserting new characters, removing characters, and replacing characters.

Inserting new characters. If the **<nDelete>** parameter is zero, then the **<cInsert>** will be inserted and the string made larger. For example,

```
? stuff("Clipper is powerful",9,0,"5 ")

// Displays "Clipper 5 is powerful"
```

Deleting characters. If the **<nDelete>** parameter is greater than zero and the **<cInsert>** string is null, characters will be removed and the string will be made shorter. For example,

```
mname := "Ms. Sandy Booth"
x := at(".",mname)
mname := stuff(mname,1,x+1)    // Produces "Sandy Booth"
x := at(" ",mname)
mname := stuff(mname,x-1,99)  //  Produces "Sandy"
```

Replacing characters. If the **<nDelete>** parameter is greater than zero, and the **<cInsert>** value is not a null, characters will be replaced in the string. The size of the new string will depend on the length of the **<cInsert>** value and the number of characters deleted. For example:

```
? stuff("DBASE III is number one!",1,9,"Clipper 5")

// Displays "Clipper 5 is number one!"
```

In this case the string size remains the same. You could also expand abbreviations using STUFF(). For example:

```
? stuff("Philadelphia, PA",15,2,"Pennsylvania")

// Displays "Philadelphia, Pennsylvania"
```

In this case the string size has increased by ten characters.

Creating and manipulating substrings

A substring is a smaller string extracted from a larger string. Clipper provides five functions to create substrings.

AT()

AT() is used to determine the starting position of a substring within a string. Its syntax is:

```
nPosition := AT(string1, string2)
```

The AT() function will search through **string2** to see if the pattern specified by **string1** occurs anywhere. If it does not appear, the function will return a zero. If the pattern does appear, the position from the beginning of the string will be returned. For example:

```
?  at( "," , "Booth, Joseph")    // returns 6
?  at( "a" , "ABCDEFG" )         // returns 0
```

AT() can be used in conjunction with the LEFT() and RIGHT() functions to produce very powerful functions. The following code illustrates how a name field can be split apart and replaced into first and last names fields. Assume the structure of the customer database has the following three fields:

```
Full_name       Char 25
First           Char 12
Last            Char 15

select CUSTOMER
go top
do while .not. eof()
   if  (x := at("," , CUSTOMER->Full_name) )  > 0
     replace CUSTOMER->First with  left(Full_name,x-1)
     replace CUSTOMER->Last  with  right(Full_name,25-x)
   endif
skip +1
enddo
```

RAT()

RAT() is used to determine the starting position of the last occurrence of a substring within a string. Its syntax is:

```
nPosition := RAT( string1, string2 )
```

The RAT() function will search through **string2** to see if the pattern specified by **string1** occurs anywhere. The search is performed in a right-to-left manner, as opposed to the left-to-right manner performed by AT(). If the substring does not appear, the function will return a zero. If the pattern does appear, the position from the beginning of the string will be returned. For example:

```
mfilename := "C:\CLIPPER5\INCLUDE\STD.CH"
x := rat("\",mfilename)
if x > 0
   ? "Path is "+left(mfilename,x-1)
endif
```

The program will display “Path is C:\CLIPPER5\INCLUDE”.

LEFT()

The LEFT() function creates a substring by extracting a number of characters from the leftmost portion of the string. Its syntax is:

```
<cSubstring> := LEFT(<cString>,<nSize>)
```

RIGHT()

The RIGHT() function creates a substring by extracting any number of characters from the rightmost portion of a string variable. Its syntax is:

```
<cSubstring>  := RIGHT( <cString>,<nSize> )
```

SUBSTR()

The SUBSTR() function creates a substring by extracting a number of characters from anywhere within the string. Its syntax is:

```
<cSubstring> := SUBSTR(<cString>,<nStart>,<nCount>)
```

The **<nStart>** parameter indicates where to start in the larger string. If **<nStart>** is positive, the starting position is relative to the beginning of the string. If **<nStart>** is negative, the starting position is relative to the end of the string.

The optional **<nCount>** parameter indicates how many characters to copy into the substring. If it's not passed, the function will extract all characters from the **<nStart>** position to the end of the string.

The example in Listing 20.3 takes a fully qualified file name and returns an array of three substrings: the drive letter, the path name, and the file name.

Listing 20.3 Split()

```
function Split(cFullname)
local x:=at(":",cFullname), y:=rat("\",cFullname)
local the_drive:="", the_path:="", the_file:=""
if x > 0
   the_drive := substr(cFullname,1,x-1)
endif
if y > 0
   the_path  := substr(cFullname,x+1,y-x)
   the_file  := substr(cFullname,y+1)
endif
return {the_drive, the_path, the_file}
```

Functions which return string values

Many of the Clipper functions return string values. Most of these functions are detailed in other chapters. Table 20.6 is a list of functions which return a string value.

Table 20.6 String functions

Function	*Description*
ALIAS()	Specified work area's alias name
CDOW()	Character day of week
CHR()	Character associated with an ASCII value
CMONTH()	Character month name
CURDIR()	The current DOS directory
DBFILTER()	Filter condition in a work area
DBRELATION()	Expression used to relate files together
DTOC()	Date in SET DATE format (MM/DD/YY)
DTOS()	Date in format YYYYMMDD
FIELD()	Name for a field number
FREAD()	Read bytes from a file using low-level function

FREADSTR()	Read a string from a file using low-level function
GETENV()	Get an environment string value
INDEXEXT()	Default index file extension, NTX or NDX
INDEXKEY()	Index key expression
I2BIN()	Integer to a binary string
L2BIN()	Long number to binary string
NETNAME()	Return the name of the network workstation
OS()	Get the operating system name
SOUNDEX()	Create a soundex code from a string
STR()	Convert a numeric variable to a string
TIME()	Current time as a string of HH:MM:SS
VERSION()	Return the Clipper version number

Useful string manipulation UDFs

To illustrate the power provided by the string tools in Clipper, we will use these tools to write some string manipulation user-defined functions.

Parse() — create an array from a string

Parse() takes a string and an optional delimiter character and returns an array which contains each token from the string. For example, the string:

```
"UPD_CUST,5,10,20,60,GR+/W"
```

would be broken into a six-element array. Each element would be of character type. The six elements are:

```
1     UPD_CUST
2     5
3     10
4     20
5     60
6     GR+/W
```

The function syntax is listed below:

```
array   := Parse( <string> [,<delimiter>] )
```

If the delimiter is not passed, it is assumed to be a comma.

Listing 20.4 contains the code for this function. It should be compiled in a separate PRG file using the /n /m compiler switches:

Listing 20.4 Parse()

```
static  def_delim := ","       // default delimiter

function Parse( pstring,pdelim )
local rlist :={}, k
pdelim := if(pdelim==nil, def_delim, pdelim)
do while len(pstring) > 0
   if (k := at(pdelim,pstring) ) > 0
      aadd(rlist,left(pstring,k-1))
      pstring := substr(pstring,k+1)
   else
      aadd(rlist,pstring)
      pstring :=""
   endif
enddo
return rlist
```

To provide an example, Listing 20.5 illustrates an application that will prompt the user for a database name to open. If the database file is not found in the current directory, the path will be searched. If found, the file will be opened, otherwise an error message will be displayed and the user will be asked to enter the file name again.

Listing 20.5 Parsedemo

```
#include "inkey.ch"
local mfile := space(8), ok := .f., k, tfile

do while ! ok
   @ 10,10 say "File Name: " get mfile pict "!!!!!!!!"
   read
   if lastkey() <> K_ESC
      tfile := trim(mfile)+".DBF"
      if ! file(tfile)
           pathlist := Parse( getenv("PATH"),";" )
           for k := 1 to len(pathlist)
              tfile := pathlist[k]+"\"+trim(mfile)+".DBF"
              if file(tfile)
                 exit
              endif
           next
      endif
      if file(tfile)
          use (tfile) new alias mfile
          ok := .t.
      else
          ? "File "+mfile-" cannot be found"
         endif
   else
         exit
   endif
enddo
```

Occurs() — count occurrences of a substring

Occurs() counts how many times a substring occurs within a larger string. Its syntax is:

```
<nCount> := Occurs(<cChar>,<cString>)
```

Here's an example of the Occurs() function:

```
function Occurs(cChar, cString)
local x:=len(cString), y:=len(strtran(cString, cChar,""))
return (x-y)/len(cChar)
```

This function illustrates a use of Clipper's STRTRAN() function. The STRTRAN() function replaces all occurrences of the sought after string with a null character. This reduces the length of the string for each occurrence. The function returns the difference in lengths between the two strings divided by the size of the string being sought.

Findany() — find first occurrence from a list

The Findany() function finds the first occurrence of a character within a character string. It is similar to the AT() function, but allows you to look for more than one possible character. Its syntax is:

```
<nPosition> := Findany(<cList>,<cSearch_string>)
```

Findany() can be used to break longer strings into tokens even if there are several possible delimiters which might be present.

Listing 20.6 Findany()

```
function Findany(chr_list, search_str)
local where:=0, k, the_char
for k:=1 to len(chr_list)
   the_char :=substr(chr_list,k,1)
   where := at(the_char,search_str)
    if where > 0
       exit
    endif
next
return where
```

Encrypt()

Data encryption is a technique to store information in a fashion that hides its meaning. It is very useful for passwords and other sensitive data in a computer file. There are many algorithms that can be used to encrypt character strings. The example listing below employs a simple substitution algorithm. While not the most secure encryption scheme, it should deter the casual user from determining the hidden information. The syntax for the function is:

```
<cEncrypted_string> := Encrypt( <cPlain_string>,<cKey> )
```

The key can be any number of characters in length. It is used to determine what letter is substituted in the password. If no key is specified the default key will be CLIPPER.

```
function Encrypt(src_word, passkey)
local pwd:="", jj, kk:=0, x
passkey:=if(passkey==nil,"CLIPPER",passkey)
for jj:=len(src_word) to 1 step -1
   kk :=if(kk>len(passkey),1,kk)
   x :=asc(substr(src_word,jj,1))+asc(substr(passkey,kk,1))
   pwd += chr(x)
next
return pwd
```

Decrypt()

Decryption is a technique that takes an encrypted string and converts it back to its original meaning. The Decrypt() function listing we show below reads the strings encrypted by the Encrypt() function as shown in the above example. Its syntax is:

```
<cPlain_string> := Decrypt( <cEncrypted_String>,<cKey> )
```

The key must be identical to the key that the password was created with. If no key is specified the default key of CLIPPER will be used.

```
function Decrypt(enc_word, passkey)
local plain:="", jj, kk:=0, x
passkey:=if(passkey==nil,"CLIPPER",passkey)
for jj:=len(enc_word) to 1 step -1
   kk :=if(kk>len(passkey),1,kk)
   x  :=asc(substr(enc_word,jj,1))- asc(substr(passkey,kk,1))
   plain += chr(x)
next
return plain
```

Summary

After reading this chapter you should be comfortable with using character strings in Clipper. You should understand the method that is used to compare strings and you should be familiar with Clipper's string manipulation functions. The user-defined functions provided as examples should also be useful in your toolbox of Clipper functions.

Reexamination 90/005,727

The Original

Page 910 of Part IV

Is Missing

Initial : AR
Date : 6-8-2000

CHAPTER 21

Math and Date Functions

Clipper provides a variety of operators for mathematical operations and comparison. Clipper also interprets dates as numeric values and allows mathematical operations to be performed on them as well.

In this chapter we will cover the functions provided by Clipper for mathematics and for date manipulation. In addition, we will write several user-defined functions for common business, statistical, and mathematical operations. Finally, we will look at the way that Clipper handles time variables.

Mathematical operators provided by Clipper

Clipper provides the mathematical operators listed in Figure 21.1. (See Chapter 5, "Operators", for a detailed discussion.)

Figure 21.1 Clipper mathematical operators

%	Modulus of two numbers
*	Multiplication
** ^	Exponentiation
+	Addition or positive sign
/	Division
-	Subtraction or negative sign
:=	Inline assignment

nVar++	Post-Increment operator
++nVar	Pre-Increment operator
nVar--	Post-Decrement operator
--nVar	Pre-Decrement operator

The operators can be used on Clipper variables, functions which return a numeric value, or numeric constants. For example,

```
nVar := 10
? nVar % 3           // Returns   1
? 10 * 15            // Returns 150
? nVar**2            // Returns 100
?--nVar              // Returns   9
? int(nVar) + 5      // Returns  15
```

When coding mathematical expressions, Clipper enforces a precedence of operators as listed in Figure 21.2. This precedence can be overridden by use of parentheses. Operations surrounded by parentheses are always evaluated first, in a left-to-right order. For example:

```
? 5 * 4 + 3        //  Returns 23
? 5 * ( 4 + 3 )    //  Returns 35
```

Figure 21.2 Operator precedence

1. Functions are always evaluated first	
2. Pre-increment, Pre-decrement	++nVar ,--nVar
3. Positive, Negative sign	+nVar , -nVar
4. Exponentiation	**, ^
5. Multiplication, division, modulus	* , / , %
6. Addition, subtraction	+ , -
7. Inline Assignment	:=
8. Post-increment, Post-decrement	nVar++ , nVar--

There is tremendous flexibility in the operators that Clipper provides. This can cause some confusion. For example, what is the result of the formulas in Listing 21.1?

Listing 21.1 Formula examples

```
? "One...: ", 10 * 15 + 5 * 6
? "Two...: ", --10 * J:=5
? "Three.: ", 5++ * 8 - 9--
```

Try to determine the results based upon the precedence table in Figure 21.2. If you are unsure of the results, create a Clipper program to see how it evaluates those expressions.

You should try to use parentheses whenever possible to prevent problems when Clipper's assumptions differ from your own. This is especially important if the formula might be modified or if the code is being maintained by another programmer. The parentheses allow explicit determination of how expressions should be evaluated.

Clipper's mathematical operators are described in more detail in Chapter 5, "Operators."

Environment setting

When mathematical operations are displayed in an unformatted form, as produced with the ? and ?? display commands, Clipper determines the appropriate number of decimal places to display. Clipper also allows the number of decimal places to be specified by the programmer using two SET commands.

SET DECIMALS

The SET DECIMALS setting determines how many decimals places should be displayed. Its syntax is:

```
SET DECIMALS TO <nValue>
```

The *<nValue>* is an integer value indicating number of decimal places. The default value is two. The default value is used if SET DECIMALS does not appear in your program. If you write SET DECIMALS TO without a parameter, the **<nValue>** is zero.

SET FIXED

The SET FIXED command tells the system how to display numeric values. If SET FIXED is ON or .t., the value of SET DECIMALS determines the exact number of decimal places to display. If SET FIXED is OFF or .f., the value of SET DECIMALS will return the minimum number of decimal places for the function or operation to display. The syntax for the command is:

```
SET FIXED  ON | OFF | <lFixed>
```

The default value is OFF, letting the default decimal places be displayed.

Listing 21.2 illustrates some examples of SET FIXED and SET DECIMALS.

Listing 21.2 SET FIXED/SET DECIMALS

```
?  1 / 3               // Displays  .33
SET DECIMALS TO 4
?  1 / 3               // Displays  .3333
SET FIXED ON
?  1 / 3               // Displays  .3333
SET DECIMALS TO 2
?  1 / 3               // Displays  .33
```

Neither SET DECIMALS nor SET FIXED affects the accuracy of the calculations. They only affect how the information is displayed. Numeric display can be controlled much more accurately by using the TRANSFORM() function or the PICTURE clause in the @..SAY command. TRANSFORM() is described in Chapter 4 and the @..SAY command is described in Chapter 13.

Math functions

In addition to the operators, Clipper provides a variety of mathematical functions. These functions give the programmer access to additional operations needed by some mathematical formulas.

ABS()

The ABS() function returns the absolute value of a number. The absolute value is the value without regard to its sign. If the number is negative, multiplying it by minus one will produce an absolute number. Its syntax is:

```
<nResult> := ABS( <nExpression> )
```

The algorithm to compute an absolute number is:

If the expression is less than zero, return -1 times the expression; otherwise, just return the value.

The ABS() function can be used to compare the difference between two numbers without regard to the sign. For example, if we subtract two dates we will have the number of days between the dates.

```
mdate1 := ctod("01/22/89")
mdate2 := ctod("02/15/89")
diff   := mdate1 - mdate2
? diff          //   Returns  -25
? abs(diff)     //   Returns   25
```

EXP()

The EXP() function returns the value of **e**x**. It is primarily used to estimate growth patterns. The approximate value of **e** is 2.71828. The syntax for EXP() is:

```
<nResult> := EXP( <nValue> )
```

The *<nValue>* represents the number which **e** should be raised to. The maximum **<nValue>** can be 45 before a numeric overflow will occur.

For example,

```
? Exp(0)       // Returns 1
? Exp(1)       // Returns 2.7182818285
? Exp(13.3)    // Returns 597,196
```

The EXP() function is the inverse of the LOG() function.

INT()

The INT() function returns the integer portion of a number. The integer portion is the value without any numbers to the right of the decimal point. Its syntax is:

```
<nResult> := INT( <nValue> )
```

The INT() function can be used to determine if a number is a multiple of another number. For example:

```
mValue := 15
? mValue/2 == int(mValue/2)   // Returns false, since 15 is
                              // not a multiple of two.
mValue := 8
? mValue/2 == int(mValue/2)   // Returns true, since 8 is a
                              // multiple of two.
```

LOG()

The LOG() function returns the natural logarithm of the number passed as a parameter. The syntax for LOG() is:

```
<nResult> := LOG( <nValue> )
```

The *<nValue>* represents the number for which the logarithm should be returned. It must be greater than zero or a numeric overflow condition will occur.

For example,

```
? Log(0)            // Returns **********
? Log(2.71828)      // Returns 1
? Log(10)           // Returns 2.30
```

The LOG() function is the inverse of the EXP() function.

MAX()

The MAX() function compares two values and returns the larger of the two. The parameters may be numbers or dates. Its syntax is:

```
<nLarger>  := MAX(<nFirst>,<nSecond>)
<dLarger>  := MAX(<dFirst>,<dSecond>)
```

The returned value will either be a date or a number, depending upon the types of the parameters.

The MAX() function can be used to determine the largest value in a database file. Listing 21.3 illustrates the use of MAX() within a DBEVAL() code block to find the highest salary in a payroll .DBF file.

Listing 21.3 MAX() Example

```
local biggest :=0
use PAYROLL new
dbeval( {|| biggest := Max(biggest,PAYROLL->Salary)} )
return biggest
```

MIN()

The MIN() function compares two values and returns the smaller of the two. The parameters may be numbers or dates. Its syntax is:

```
<nSmaller>  := MIN(<nFirst>,<nSecond>)
<dSmaller>  := MIN(<dFirst>,<dSecond>)
```

The returned value will either be a date or a number, depending upon the types of the parameters.

The MIN() function can be used to determine the earliest date in a database file. Listing 21.4 illustrates the use of MIN() within a DBEVAL() code block to find the earliest invoice date in a transaction file.

Listing 21.4 MIN() Example

```
local earliest := ctod("12/31/2199")
use TRANS new
dbeval( {|| earliest := min(earliest,TRANS->inv_date)} )
return earliest
```

ROUND()

The ROUND() function returns a number rounded to the specified number of digits. Its syntax is:

```
<nResult>  := ROUND( <nValue>,<nDecimals> )
```

<nValue> represents the number to be rounded. ***<nDecimals>*** indicates the number of decimal places to be retained. If **<nDecimals>** is a negative number, whole digits will be rounded.

For example:

```
? Round(100.1245,2)     // Returns   100.12
? Round(541.12965,3)    // Returns   541.130
? Round(254.691,0)      // Returns   255
? Round(256,-1)         // Returns   260
? Round(254,-1)         // Returns   250
```

If the first digit past the rounding is greater than or equal to five, the number will be rounded up. If the number is less than five, the number will be rounded down. Also keep in mind that ROUND() actually changes the value of the number. TRANSFORM() is used to display numbers without changing their values.

SQRT()

The SQRT() function returns the square root of a number. The syntax for Clipper to calculate a square root is:

```
<nRoot>  := SQRT( <nValue> )
```

<nValue> is the number of which the square root should be taken. ***<nRoot>*** is the result, the square root of the number. For example, Listing 21.5 lists the square roots of all numbers from one through ten.

Listing 21.5 Square root example

```
local jj
for jj:= 1 to 10
   ? jj, sqrt(jj)
next
```

Business user-defined functions

The world of business requires many specialized functions which do not fit in the syntax of a programming language, yet are frequently needed in many applications. In this section, we will provide functions for depreciation and interest calculations.

Depreciation

When a business invests money in capital equipment, it is generally expected to last several years. Since the benefit of using the equipment is spread out, the expense of acquiring it should be spread out over the same period of time. Depreciation is the method to attempt to spread the invested money out over several years.

To determine each period's depreciation amount, we need first to identify some terms. Table 21.1 defines common terms for depreciation.

Table 21.1 Depreciation Terms

Acquisition Cost	The purchase price, plus any shipping incurred, that the asset cost.
Useful life	A projected number of years (or months) over which the asset is expected to be useful.
Scrap value	The expected price for which the asset can be sold after it has passed its useful life.
Accumulated	Running total of all depreciation taken to depreciation date on an asset.
Book Value	The acquisition cost of the asset minus its accumulated depreciation.

Straight-line. Straight-line depreciation assumes that the asset value is evenly divided over its useful life. This method is the simplest depreciation to calculate, but also makes no provision for inflation.

The syntax for Sl() is:

```
<nArray> := Sl( <nCost>,<nScrap>,<nLife> )
```

The function will return an array filled with each period's depreciation amount.

Sum of the Year's Digits. One of the drawbacks of the straight-line method is that it makes no allowances for inflation. The sum of the year's digits depreciation method improves upon straight-line by taking more of a depreciation expense in the early years and less later.

The annual depreciation entry for sum of the year's digits is computed by summing up the digits of the year and computing a percentage based on the years of life left over total digits. For example, assume an asset has a life of five years and is worth $3000 with no scrap value. Figure 21.4 shows how the sum of the year's digits depreciation amounts would be calculated.

Figure 21.4 SYD Depreciation

(a)	Sum all years digit's...:		1+2+3+4+5	= 15	
(b)	Year one	5 years left:	5/15 = 33%	or	$1,000
	Year two	4 years left:	4/15 = 27%	or	800
	Year three	3 years left:	3/15 = 20%	or	600
	Year four	2 years left:	2/15 = 13%	or	400
	Year five	1 year left.:	1/15 = 7%	or	200

The sum of the year's digits is referred to as an accelerated depreciation formula, since more depreciation is taken in the earlier years of the asset's life.

The syntax for the Syd() function is:

```
<nArray> := Syd( <nCost>,<nScrap>,<nLife> )
```

The function will return an array filled with each period's depreciation amount.

Declining Balance. Another accelerated depreciation method is the declining balance method. This method uses a higher percentage on a progressively smaller asset value. This results in taking more depreciation expense in the early years and less later.

The annual depreciation entry for the declining balance method is computed by applying a percentage to the asset's book value, not its acquisition cost. The book value will get lower after each depreciation entry is recorded, but the percentage to depreciate remains the same. For example, assume we purchased an asset for $5,895; it has a scrap value of $1,500. Figure 21.5 shows how the declining balance depreciation amounts would be calculated.

Figure 21.5 Declining balance depreciation

			Ending
	Balance	*Depreciation*	*Balance*
Year one	5,895.00	1,411.62	4,483.38
Year two	4,483.38	1,073.59	3,409.78
Year three	3,409.78	816.51	2,593.27
Year four	2,593.27	620.99	1,972.29
Year five	1,972.29	472.29	1,500.00

The syntax for the Db() function is:

```
<nArray> := Db( <nCost>,<nScrap>,<nLife> )
```

The function will return an array filled with each period's depreciation amount.

Double-declining Balance. Another accelerated depreciation method is the declining balance method. This method uses a higher percentage on a progressively smaller asset value. This results in taking more depreciation expenses in the early years and less later.

The annual depreciation entry for the declining balance method is computed by applying a percentage to the asset's book value, not its acquisition cost. The book value will get lower after each depreciation entry is recorded, but the percentage to depreciate remains the same. For example, assume we purchased an asset for $1,000; it has a scrap value of $100 and a seven-year life. Figure 21.6 shows how the declining balance depreciation amounts would be calculated.

Note that during the second half of the asset's life, the formula switches over to straight-line depreciation.

Figure 21.6 Double Declining Balance Depreciation

	Balance	*Depreciation*	*Ending Balance*
Year one	1,000.00	257.14	742.86
Year two	742.86	183.67	559.18
Year three	559.18	131.20	427.99
Year four	427.99	93.72	334.27
Year five	334.27	78.09	256.18
Year six	256.18	78.09	178.09
Year seven	178.09	78.09	100.00

The syntax for the Ddb() function is:

```
<nArray> := Ddb( <nCost>,<nScrap>,<nLife> )
```

The function will return an array filled with each period's depreciation amount.

Listing 21.6 contains the four depreciation user-defined functions.

Listing 21.6 Depreciation

```
function Sl(nCost, nScrap, nLife)
local retarray[nLife],dp_amt := (nCost-nScrap)/nLife
afill(retarray,dp_amt)
return retarray
*
function Syd(nCost, nScrap, nLife)
local jj, arr_:={}
for jj:=1 to nLife
   aadd(arr_,2*(nCost-nScrap) * (nLife+1-jj)/(nLife*(nLife+1)))
next
return arr_
*
function Db(nCost, nScrap, nLife)
local accum:=0, jj, arr_:={}
for jj:=1 to nLife
   aadd(arr_,(1-(nScrap/nCost) ** (1/nLife)) * (nCost-accum))
   accum += arr_[jj]
next
return arr_
*
function Ddb(nCost, nScrap, nLife)
local accum:=0, jj, arr_:={}, factor:=2/nLife
local half:=int((nLife+1)/2)+1
for jj:=1 to nLife
   if jj < half
     aadd(arr_,factor*(nCost-nScrap-accum))
   elseif jj==half
     aadd(arr_,(nCost-nScrap-accum)/(nLife+1-half))
   else
     aadd(arr_,arr_[jj-1])
   endif
   accum += arr_[jj]
next
return arr_
```

Interest

When Benjamin Franklin said "Time is money," he seemed to be predicting the future of banking and finance. Money and time may have an adversarial or beneficial relationship, depending upon whether you are borrowing it or saving it. The functions in Listing 21.7 provide interest calculations and time value tools.

Interest calculations require the definition of several terms. These are listed in Table 21.2.

Table 21.2 Interest Terms

Principal	Initial balance or investment amount.
Rate	Percentage rate of interest payment. For purpose of these UDFs, the rate should be expressed as a decimal value (i.e. 10% = .10).
Periods	Number of periods (usually years) for calculations to use.
Payment	Regular, periodic payment.

Simple Interest. Simple interest is a payment based upon the original principal, without taking into account interest previously earned. For example, assume you opened a savings account at 6% simple interest with a $1,000 deposit. A year later that account should have $1,060 in it. (Don't forget to report the $60 to the IRS!)

The Simple() function in Listing 21.7 computes simple interest. Its syntax is:

```
<nFinal_value> := Simple(<nPrincipal>,<nRate>,<nPeriods>)
```

The ***<nFinal_value>*** returned shows how much principal and interest will be accumulated at the end of the term of **<nPeriods>** .

Compound Interest. Compound interest is a payment based upon the original principal, plus any interest earned to date. Depending upon how frequently the interest is added to the balance, the effective interest rate will be higher than it would be under simple interest. Using our same example, assume you made that deposit in an account that compounds interest quarterly. A year later that account should have $1,060 in it. Figure 21.7 shows how this number was derived.

Figure 21.7 Compound interest on $1,000

Quarter	Interest	Balance	$1,000
1	$ 15.00	$ 1,015.00	
2	$ 15.23	$ 1,030.23	
3	$ 15.45	$ 1,045.68	
4	$ 15.68	$ 1,061.36	
		$ 1,061.36	

The Compound() function in Listing 21.7 computes compound interest. Its syntax is:

```
<nValue> := Compound(<nPrin>,<nRate>,<nPeriods>,<cFreq>)
```

The ***<nValue>*** returned shows how much principal and interest will be accumulated at the end of the term of **<nPeriods>** assuming a **<cFreq>** of one of the following: (A)nnually, (Q)uarterly, (M)onthly, or (C)ontinuously.

Present Value. Present value determines the current value today of a stream of future payments. The formula to compute present value is shown in Figure 21.8.

Figure 21.8 Present value formula

$$\text{Payment} \quad * \quad \frac{1 - (\,1 + \text{Rate}\,)^{(\text{Periods}*-1)}}{\text{Rate}}$$

The Pv() user-defined function in Listing 21.7 computes present value. Its syntax is:

```
<nPresent> := Pv(<nPayment >,<nRate>,<nPeriods>)
```

<nPresent> contains the amount that represents today's value of the investment. ***<nRate>*** should be expressed as a fraction. For example, if you were to win a million dollars in the lottery and be guaranteed $50,000 a year for the next twenty years, assuming 12% yearly interest the present value of your winnings is $373,472.18.

```
? Pv(50000,.12,20)   // would display  373472.18
```

Future Value. Future value determines how much money will accumulate at the end of a period of time, assuming you make regular payments. The formula to compute future value is shown in Figure 21.9.

Figure 21.9 Future value formula

$$\text{Payment} \quad * \quad \frac{(1 + \text{Rate})^{\text{Periods}} - 1}{\text{Rate}}$$

The Fv() user-defined function in Listing 21.7 computes future value. Its syntax is:

```
<nFuture> := Fv(<nPayment>,<nRate>,<nPeriods>)
```

<nFuture> contains the amount that will be accumulated at the end of the period. ***<nRate>*** should be expressed as a fraction. For example, if you were to deposit $500 once a year in an investment paying 15% interest for a period of six years, you would have $4,376.87 at the end of the sixth year.

```
? Fv(500,.15,6)   // would display 4,376.87
```

Listing 21.7 Interest

```
function Simple(nPrin, nRate, nPeriods)
return nPrin * (nRate*nPeriods)
*
function Compound(nPrin, nRate, nPeriods, cFreq)
local jj,arr_ := {},multiplier := 1
cFreq := if(cFreq==nil,"Q",cFreq)
if cFreq == "C"
   multiplier := Exp(nRate)
elseif cFreq == "A"
   multiplier := (1+nRate)
elseif cFreq == "Q"
   multiplier := (1+nRate/4)**4
elseif cFreq == "M"
   multiplier := (1+nRate/12)**12
endif
aadd(arr_, nPrin * multiplier)
for jj := 2 to nPeriods
   aadd(arr_, arr_[jj-1] * multiplier)
next
return arr_
*
function Pv(payment, rate, periods)
local multi:= (1-((1+Rate)**(-1*periods))) / rate
return multi * payment
*
function Fv(payment, rate, periods)
local multi:= (((1+Rate)**periods)-1) / rate
return multi * payment
```

Statistical user-defined functions

Listing 21.8 includes several user-defined functions to calculate combinations and permutations, to calculate group statistics, and to create pseudo-random samples and normalized numbers.

Combo()
Combinations represent the number of possible groups of items that can be extracted from a larger group. The order of items in the groups is not important. A good example is to determine the number of five card poker hands in a deck of 52 cards. The Combo() function syntax is:

```
<nCombinations> := Combo( <nGroup>,<nSets> )
```

<nGroup> represents the total number of items in the larger group. ***<nSets>*** represents the number of items in each set or smaller group.

For example:

```
? Combo( 52,5 )   // Displays 259894
```

Perms()
Permutations represent the number of possible arrangements that can be extracted from a larger group. The order of items in the groups is important, since each different order represents a new arrangement. The Perms() function syntax is:

```
<nPermutations> := Perms( <nGroup>,<nArrange> )
```

<nGroup> represents the total number of items in the larger group. ***<nArrange>*** indicates the number of items in each arrangement.

For example:

```
? Combo( 10,3 )   // Displays 720
```

Mean()
The population mean is a statistical term that denotes the average value of a group of data. The Mean() function in Listing 21.8 computes the population mean for an array. Its syntax is:

```
<nMean> := Mean( <aGroups> )
```

<aGroups> is a numeric array containing the values to be averaged. For example:

```
local groups := {}, mean := 0
select PAYROLL
go top
do while !eof()
   aadd( groups, PAYROLL->salary )
   skip +1
enddo
mean := mean( groups )
```

Median()

The median is a statistical term that is used to measure central tendency in a group of data. The mean can be misleading, since it is possible that no element in the group has the mean value. For example, if two people's heights are six feet and five feet, the mean height is five foot six inches, even though no members of the group are that height. The median represents the middle value from the group, which generally means at least one member of the group is the median value. The Median() function in Listing 21.8 computes the median for an array. Its syntax is:

```
<nMedian> := Median( <aGroups> )
```

<aGroups> is a numeric array containing the values to be used. For example:

```
local groups := {}, median := 0
select PAYROLL
go top
do while !eof()
   aadd( groups, PAYROLL->salary )
   skip +1
enddo
median := Median( groups )
```

Var()

The variance is a measure of how far values are from the midpoint of the group. It is used to determine the distribution of values. The Var() function in Listing 21.8 computes the variance of an array of numbers. Its syntax is:

```
<nVariance>  := Var( <aGroups> )
```

<aGroups> is a numeric array containing the values to be computed. For example:

```
local groups := {}, variance := 0
select PAYROLL
go top
do while !eof()
   aadd( groups, PAYROLL->Salary )
   skip +1
enddo
variance := Var( groups )
```

Listing 21.8 Statistical functions

```
function Combo(nNumber,nTaken)
local nLogn := factorial(nNumber), nLogt := factorial(nTaken)
local nDiff := factorial(nNumber-nTaken)
return int(exp(nLogn-(nLogt+nDiff))+.5)

function Perms(nNumber,nTaken)
local nLogn := factorial(nNumber), nLogt := factorial(nTaken)
return int(exp(nLogn-nLogt) + .5)

static function Factorial(nX)
local appx := 1,jj
if nX <=0
   appx := 0
else
   for jj := 1 to 10
      appx *= jj
```

```
         if nX == jj
            return log(appx)
         endif
      next
      /* Sterling approximation for large integers */
      appx := log(6.283186)/2+log(nX) * (nX+.5)-nX+1/(nX*12)
   endif
   return appx

   function Mean(aGroups)
   local nSum := 0,nSize := len(aGroups),jj
   for jj := 1 to nSize
      nSum =+ aGroups[jj]
   next
   return if(nSize==0,0,nSum/nSize)

   function Median(aGroups)
   local nSize := len(aGroups), nMid, nMedian
   local lOdd :=  (nSize/2 <> int(nSize/2))
   asort(aGroups,1,nSize)
   if lOdd
      nMedian := aGroups( int(nSize/2)+1 )
   else
      nMid    := int(nSize/2)
     nMedian := (aGroups(nMid)+aGroups(nMid+1))/2
   endif
   return nMedian

   function Variance(aGroups)
   local nSum := 0, nSum_Square := 0, jj, nSize := len(aGroups)
   for jj := 1 to nSize
      nSum += aGroups[jj]
     nSum_Square += (aGroups[jj]**2)
   next
   return nSum_square/nSize-(nSum/nSize)**2
```

Mathematical user-defined functions

Listing 21.11 includes several user-defined functions to convert numbers between bases and to calculate prime numbers, greatest common divisors, and least common multiples.

Bin2Dec()/Dec2Bin()

Base two or binary numbers are representations of numbers using only the digits zero and one. The Bin2Dec() function reads a binary string and returns a decimal number. Its syntax is:

```
<nValue>  := Bin2dec( <cString_of_zeros_and_ones> )
```

A decimal number may also be converted into base two. The Dec2Bin() function reads a decimal number and returns a binary string. Its syntax is:

```
<cString_of_zeros_and_one> := Dec2Bin(<nDecimal> )
```

These functions can be used to work with bit strings in Clipper. For example, DOS stores the computer's equipment list as a bit string as illustrated in Table 21.1. Using the FT_PEEK() function written by Ted Means, the code in Listing 21.9 returns an array containing the equipment list of the computer.

Table 21.1 DOS Equipment List Byte

Bits	*Contents*
0	1 - Disk drives are present, 0 - no disk drives
1	Not used, should be 0
2-3	System RAM 00=16K, 01=32K, 11=64K
4-5	Video mode 01=40 Columns 10=80 Color 11=80 B&W
6-7	Diskette drives 00=1, 01=2, 10=3, 11=4 drives
8	DMA installed? 1=Yes, 0=No
9-11	Number of serial ports
12	Game adapter 1=Yes, 0=No
13	Serial printer installed 1=Yes, 0=No
14-15	Number of printers

Listing 21.9 Read equipment list

```
function Equip
local eByte1 := Ft_peek(0,272)
local eByte2 := Ft_peek(0,273)
local bString1 := "", bString2 := ""
local arr_ := {}, cSport
bString1 := padl(Dec2Bin(eByte1), 8, "0")
bString2 := padl(Dec2Bin(eByte2), 8, "0")
cSport := substr(bstring2, 7, 1) +; substr(bstring2, 6, 1)
aadd(arr_, str(Bin2Dec(cSport), 2)+;" Serial Ports")
aadd(arr_, if(substr(bstring2, 4, 1) == "1", "", "No ") +;
         "Game Port")
aadd(arr_, if(substr(bstring2, 8, 1) == "1", "", "No ") +;
         "DMA Chip")
return arr_
```

Hex2Dec()/Dec2Hex()

Base sixteen or hexidecimal numbers are representations of numbers using only the digits zero through F. (A is 10, B is 11, and so on up to F is 15.) The Hex2Dec() function reads a hexidecimal string and returns a decimal number. Its syntax is:

```
<nValue>  := Hex2dec( <cHex_string> )
```

A decimal number may also be converted into hexidecimal. The Dec2Hex() function reads a decimal number and returns a hexidecimal string. Its syntax is:

```
<cHex_string> := Dec2Hex(<nDecimal> )
```

Frequently, DOS files are viewed as hexidecimal numbers. The function in Listing 21.10 reads a DOS file and displays it as a series of hexidecimal digits. The function syntax is:

```
Hexdump( <cFilename> )
```

Listing 21.10 Hexdump

```
#define BUF_SIZE  200

function Hexdump(cFilename)
local fh := fopen(cFilename)
local buf:= space(BUF_SIZE), spot, jj, tt
if fh > -1
   do while fread(fh, @buf, BUF_SIZE) == BUF_SIZE
        for jj := 1 to 10
            ?
            for tt := 1 to 20
              spot := (jj-1) * 20 + tt
              ?? dec2hex(asc(substr(buf,spot,1)))+" "
             next
        next
    enddo
    jj := 0
    ?
  do while !empty(buf)
        jj++
        ?? Dec2Hex(asc(substr(buf,1,1))) + " "
        buf := substr(buf,2)
        if jj == 20
            jj := 0
            ?
        endif
    enddo
endif
return nil
```

Gcd()

The greatest common divisor is the largest integer that goes into two other integers. For example, the greatest common divisor of ten and fifteen is five. Euclid devised an algorithm to determine the greatest common divisor in approximately 300 B.C.

The syntax for the Gcd() function is:

```
<nGcd> := Gcd( <nFirst>,<nSecond> )
```

For example:

```
? Gcd(24,56)        // Displays 8
? Gcd(1095,155)     // Displays 5
```

Isprime()

The Isprime() function returns a logical value indicating whether or not a number is a prime number. A prime number is a number which can be evenly divided by only itself and one. The syntax for the Isprime() function is:

```
<logical>  := Isprime( <nInteger> )
```

For example,

```
local jj
for jj := 1 to 50
   if Isprime(jj)
      ? str(jj,3)," is a prime number"
   endif
next
```

Lcm()

The least common multiple is an integer value which represents the smaller number that two integers can be evenly divided into. The syntax for Lcm() is:

```
<nLeast_common_multi> := Lcm( <nInteger1>,<nInteger2> )
```

For example,

```
? Lcm( 24,56 )        // Displays 168
? Lcm( 4096,128)      // Displays 4096
```

Listing 21.11 Mathematics

```
function Bin2dec(cBinary)
local nDec := 0,jj
for jj := 1 to len(cBinary)
  nDec := nDec + nDec + val(substr(cBinary,jj,1))
next
return nDec

function Dec2Bin(nDecimal)
local nTemp := nDecimal,cBinary := ""
local nDiv  := int(nTemp/2),nRem := 0
do while nDiv <> 0
   nRem    := nTemp-(2*nDiv)
   cBinary := substr("01",nRem + 1,1) + cBinary
   nTemp   := nDiv
   nDiv    := int(nTemp/2)
enddo
return cBinary

function Hex2dec(cHex)
local nDec := 0,jj
for jj := 1 to len(cHex)
  nDec := nDec * 16 + at(substr(cHex,jj,1),"0123456789ABCDEF")
next
return nDec

function Dec2Hex(nDec)
local nTemp := nDec,cHex := ""
local nDiv  := int(nTemp/16),nRem := 0
do while nDiv <> 0
   nRem  := nTemp-(16*nDiv)
   cHex  := substr("0123456789ABCDEF",nRem+1,1) + cHex
```

```
    nTemp := nDiv
    nDiv  := int(nTemp/16)
 enddo
 return cHex

 function Gcd(nX,nY)
 local rmd := 1,quot := 0
 do while rmd <> 0
      quot := int(nX/nY)
      rmd := nX - (nY*quot)
      if rmd <> 0
        nX := nY
        nY := rmd
      endif
 enddo
 return nY

 function Isprime(nX)
 local jj
 if nX == 1
    return .F.
 elseif nX < 4
    return .T.
 endif
 if nX == int(nX/2) * 2
    return .F.
 endif
 for jj := 3 to int(sqrt(nX)) step 2
    if nX == int(nX/jj) * jj
       return .F.
    endif
 next
 return .T.

 function Lcm(nX,nY)
 return nX*nY/gcd(nX,nY)
```

Clipper dates

Clipper includes very powerful date manipulation capabilities. Dates are stored internally in such a way that math operations can be performed on dates to derive other dates. Adding an integer to a date will result in a future date. Subtracting two dates will result in the number of days between the two.

Environment setting

Clipper provides three settings which control the display of dates. One controls whether or not the century is displayed, the second controls the format of the date, and the third controls which century two-digit years are placed into.

SET CENTURY

The SET CENTURY command determines whether or not the year portion of a date is display with four digits (including the century) or two digits (not including the century). The default is two digits. The syntax of the command is:

```
SET CENTURY <ON|OFF|<logical>
```

If the setting is ON or .t., then the year is displayed using four digits. If the setting is OFF or .f., only two digits are used to display the year. The default setting is OFF.

The value of the SET CENTURY setting only affects the input and display of dates and does not affect the way a date is stored in memory or on disk.

SET DATE

Clipper supports several different date display formats. The SET DATE command is used to specify which format Clipper should display dates in. The syntax is:

```
SET DATE <literal value>
```

The literal value must be one of those formats listed in Table 21.2.

Table 21.2 SET DATE Values

American	mm/dd/yy
Ansi	yy.mm.dd
British	dd/mm/yy
French	dd/mm/yy
German	dd.mm.yy
Italian	dd-mm-yy
Japan	yy/mm/dd
USA	mm-dd-yy

You cannot use a character expression in the SET DATE command although you can macro-expand the value. For example,

```
cDformat := "AMERICAN"
set date cDformat           // Not ok
set date (cDformat)         // Still not ok!
set date &cDformat.         // This will work
```

You can also set the date format using the SET DATE FORMAT command. One benefit of the date format syntax is that it allows a character string to be used to determine the date format, rather than a literal value. The syntax for SET DATE FORMAT is:

```
SET DATE FORMAT TO <cFormat_string>
```

The format string is a combination of the following characters

YYYY	Four digit year
YY	Two digit year
MM	Month number
DD	Day number

and a separation character, such as a colon (:), a period (.), etc. The format string must be twelve or fewer characters. Clipper maps the date components onto the appropriate character, i.e. the year is mapped to YYYY, the month to MM, and the day to DD. For example:

```
set date format to "YYYY-MM-DD"
? date()                         // Displays 1991-01-22
mvar := "MM|DD/YY"
set date format to mvar
? date()                         // Displays 01|22/91
```

The SET DATE FORMAT command allows any structure of date to be used as the default date format. You may also store the date format in a memory variable to allow different users to modify their preferred date display.

SET EPOCH

The SET EPOCH command informs the system how to handle dates that use only two digits for the year. When a two-digit year is entered into a date, its year digits are compared with the year digits of the epoch setting to determine the century to place the date into. If the two digits are prior to the setting of SET EPOCH, the year is assumed to be in the next century. If the digits are greater than or equal to the SET EPOCH setting, the year is assumed to be in the current century. The setting's syntax is:

```
SET EPOCH TO <nYear>
```

The ***<nYear>*** is a four-digit year, which defaults to 1900. This default forces any date entered to be considered a date in the twentieth century.

For example,

```
set epoch to 1900
mdate := ctod("01/22/89")
? year(mdate)       // displays  1989

set epoch to 1990
mdate := ctod("01/22/89")
? year(mdate)       // displays  2089
```

A four-digit year may always be specified to explicitly determine the century that year belongs to.

Date functions

Clipper provides a large number of functions to extract components of date variables, and to convert dates to string formats. This section covers those functions.

CDOW()

The CDOW() function takes a date variable and returns the day of the week as a character string. Its syntax is:

```
<cDayname>  := CDOW( <dValue> )
```

For example, we can use the CDOW() function to display a date variable in a longer fashion.

```
mdate := ctod("01/22/89")
? cdow(mdate),mdate      // Displays  Sunday 01/22/89
```

CMONTH()

The CMONTH() function takes a date variable and returns the month of the year as a character string. Its syntax is:

```
<cMonthname>  := CMONTH( <dValue> )
```

As an example, we can use the CMONTH() function to display two dates as a range of months.

```
mdate1 := ctod("06/05/90")
mdate2 := ctod("09/15/90")
? left(cmonth(mdate1),3) + "-" + left(cmonth(mdate2),3)
```

This example will display:

```
Jun-Sep
```

CTOD()

This function reads a character string and attempts to map it into the current date format. If the string is successfully mapped, a date variable will be returned. Its syntax is:

```
<dVariable> := CTOD(<cFormatted_date_string>)
```

It is important to be aware of the current SET DATE value when using CTOD(). For example:

```
? ctod("01/05/91") // Produces 01/5/91 or January 5, 1991
                   // if SET DATE is AMERICAN and produces
                   // 05.01.91 or May 1,1991 is the SET
                   // DATE format is BRITISH or GERMAN
```

DATE()

The DATE() function returns the system date as a date variable. Its syntax is:

```
<dVariable>  := DATE()
```

For example,

```
@ 02,70 say date()
```

DAY()
The DAY() function takes a date value and returns an integer which corresponds to the day of the month. Its syntax is:

```
<nDay_of_month>  := DAY( <dValue> )
```

<dValue> must be a valid date variable. The returned value will be a number between one and thirty-one, depending upon the numbers of days in the month.

DOW()
The DOW() function takes a date value and returns a numeric value which corresponds to the day of the week. Its syntax is:

```
<nDay_number>  := DOW( <dValue> )
```

<dValue> must be a valid date variable. The returned value will be a number between one and seven. One represents Sunday and seven represents Saturday. If an empty date is passed, zero will be returned.

DTOC()
The date to character function takes a date variable as a parameter and returns a string representation of the date. The string is created in the format specified by the SET DATE or the SET DATE FORMAT command. If SET DATE has not been specified, the default date format is mm/dd/yy. The syntax for DTOC() is:

```
<cString> := DTOC( <dValue> )
```

<dValue> must be a valid date variable. If an empty date is passed, a null string will be returned.

DTOC() is frequently used to display dates. Since it formats the dates according to the SET DATE value, your program can easily work with character dates in international display formats.

DTOS()

The date to string function takes a date variable as a parameter and returns a string in the format:

YYYYMMDD

where	YYYY	represents the four digit year
	MM	represents the numeric month
	DD	represents the day of the month

The return format of the DTOS() function is not affected by the SET DATE setting. It is this string representation of the date that should be used for sorting and indexing dates, since the dates will appear in chronological order. The syntax for DTOS() is:

```
<cString> := DTOS( <dValue> )
```

<dValue> must be a valid date variable. The returned string can be used to arrange dates in chronological sequence, regardless of the date format. It is very useful when indexing databases in date order. For example, you could use DTOS() to create an index which will cause the log entries in a database to appear in chronological date sequence.

```
index on dtos(LOG_DATE) to cust01
```

The DTOS() function can also be used to create concatenated index strings. For example:

```
index on dtos(Sale_date)+state to cust01
```

MONTH()

The month function returns a two-digit month number from a date variable. Its syntax is:

```
<nMonth_no>  := MONTH( <dValue> )
```

<dValue> must be a valid date variable. The returned value will be a number between one and twelve, depending upon the month. If an empty date is passed, zero will be returned.

YEAR()

The YEAR() function takes a date value and returns a four-digit integer which corresponds to the year. Its syntax is:

```
<nYear>  := YEAR( <dValue> )
```

The YEAR() function is not affected by the SET DATE FORMAT or by the SET CENTURY setting. It will always return a four-digit year from a valid date. From an empty date, it will return a zero.

When trying to extract the year, month, or day from a date variable, it is more reliable to use the DAY(), MONTH(), and YEAR() functions rather than relying upon SUBSTR() to extract the components. Since the date format can be changed, the SUBSTR() function may not always be aware of the positioning of the components. In the AMERICAN date format, the month is the first two positions. In the GERMAN date format, the month is in positions four and five.

User-defined date functions

In addition to the standard Clipper date functions, we have included three useful user-defined functions which work with date variables.

Isleap()

The Isleap() function takes a date or a four-digit year and returns a logical value indicating whether or not the year in question is a leap year. Its syntax is:

```
<logical> := Isleap( <dValue|nYear> )
```

Isleap() works with either a four-digit numeric year or a date variable. Listing 21.12 contains the code for the function.

Listing 21.12 Isleap()

```
function Isleap(param)
local nyy := if(valtype(param) == "D", year(param),param)
local dtest := ctod("02/28/"+str(nyy,4))
return (month(dtest+1) == 2)
```

Juld()

A julian date is a three-digit integer which indicates the day of the year. The Juld() function returns a julian date for a date variable. Its syntax is:

```
<nInteger> := Juld( <dVariable> )
```

The code for the Juld date function is shown in Listing 21.13.

Listing 21.13 Juld()

```
function Juld(dVar)
local yy := str(year(dVar),4)       // Determine year
local temp := ctod("01/01/"+yy)     // Build January first of year
return (dVar-temp)                  // Subtract to return # of days
```

Djul()

The djul() function takes the three digit integer which indicates the day of the year and returns a date variable from it. Its syntax is:

```
<dVariable> := Dulj( <dVariable> )
```

The code for the Djul date function is shown in Listing 21.14.

Listing 21.14 Djul()

```
function Djul(nJulian)
local yy := year( date())
local temp := ctod("01/01/"+yy)     // Build January first of year
return (temp + nJulian)             // Add julian days back in
```

Making a calendar

To illustrate the date functions, Listing 21.15 contains a function which will display a calendar for the month. The syntax for the function is:

```
Disp_cal(<dVariable>)
```

The ***<dVariable>*** parameter must be a valid date.

Listing 21.15 Disp_cal()

```
function Disp_cal(param)
local jj,tt,temp :=str(month(param),2)+"/01/"+str(year(param),4)
local start := dow(ctod(temp)),pday:=0
local temp1 :=str(month(param)+1,2)+"/01/"+str(year(param),4)
local last := day(ctod(temp1)-1)
```

```
? " Sun Mon Tue Wed Thu Fri Sat"
?
for jj := 1 to 6
   for tt := 1 to 7
      if (tt < start+1) .and. jj == 1 .or. pday > last
         ?? space(4)
      else
         pday++
         ?? str(pday,4)
      endif
   next
   ?
next
return nil
```

Time functions

Clipper provides two functions to determine the system time. One function returns the time in a formatted fashion, while the other returns it as an integer number of seconds.

TIME()

The TIME() function returns the system time in the format of **hh:mm:ss.** **hh** is the hour past midnight based on a twenty-four hour clock. **mm** is minutes past the hour and **ss** is seconds past the minute. The TIME() function can be used to display the time on the screen or in a printed report. The code example below displays the date and time in the upper right-hand corner of the screen:

```
@ 1,70 say time()
@ 2,70 say date()
```

The syntax for TIME() is:

```
<cTime> := TIME()
```

<cTime> is always returned based upon a 24-hour clock. The AmPm() UDF in Listing 21.16 converts a time variable to a 12-hour clock format. Its syntax is:

```
<cTime12>  := AmPm( time() )
```

Listing 21.16 - AmPm()

```
function AmPm( cTime )
if val(cTime) < 12
   cTime += " am"
elseif val(cTime) == 12
   cTime += " pm"
else
   cTime := str(val(cTime) - 12, 2) + substr(cTime, 3) + " pm"
endif
return cTime
```

SECONDS()

The SECONDS() function returns the system time as an integer number of seconds past midnight. The range can be zero through 86,399 (the number of seconds in a day). Time returned as seconds is useful for billing calculations. The syntax of SECONDS() is:

```
<nSecs> := SECONDS()
```

The SECONDS() function could be used to track billable time spent on a project or a phone call. The start time and end time would both be stored as seconds past midnight. The Sec2hours function in Listing 21.17 takes a number of seconds and returns the hours and fractional hours the seconds represent. Its syntax is:

```
<nHours> := Sec2hours( <nSeconds> )
```

Listing 21.17 Sec2hours()

```
function Sec2hours(nSeconds)
local hh:= int(nSeconds/3600), mm := nSeconds % 3600, ss := 0
mm :=  int(mm/60)
ss :=  mm % 60
return hh+(mm/100)
```

User-defined functions

It is unfortunate that the Clipper time processing is not as robust as its capabilities with dates. Additionally, Clipper's time functions provide no methods for dealing with times across two dates. Processing time values requires some additional user-defined functions to round out the time processing functionality. The code for the following user-defined functions is presented in Listing 21.18.

In order to provide a more robust time environment, let's borrow the internal date and time representation from Lotus. In this format, the date is an integer value, some number of days past a base date. The time is the fractional portion of the number, indicating number of seconds past midnight of the specified date.

For example, assuming a base date of January 1, 1980:

```
3309.17400    - January 22, 1989 4:50am
```

The use of this format allows us to construct user-defined functions to manipulate times in a more powerful fashion.

DateTime()

This function takes a date and a time string and returns a numeric value that represents the date and time in the format used by Lotus. Its syntax is:

```
<nDateTime> := DateTime( <dVar>,<cTime> )
```

<dVar> must be a valid date variable and ***<cTime>*** must be a valid time string in 24 hour format. If either parameter is missing, the function will return a -1.

GetDate()
This function takes a numeric value created by the DateTime() function and returns a date variable. Its syntax is:

```
<dVariable> := GetDate( <nDatetime> )
```

If the ***<nDatetime>*** is an invalid date/time number, an empty date will be returned from the function.

GetTime()
This function takes a numeric value created by the DateTime() function and returns a formatted time string. Its syntax is:

```
<cTime> := GetTime( <nDatetime> )
```

If the ***<nDateTime>*** is an invalid date/time number, a null string will be returned from the function.

Elap_days()
This function takes two date/time values and returns the number of elapsed days between them. Its syntax is:

```
<nDays> := Elap_days( <nDateTime1>,<nDateTime2> )
```

If either ***<nDatetime>*** is invalid, the function will return a -1. If the number of hours remaining after all days are accounted for exceeds 12, then an extra day will be added to the return value.

Elap_hours()

This function takes two date/time values and returns the number of elapsed hours between them. Its syntax is:

```
<nHours> := Elap_hours( <nDateTime1>,<nDateTime2> )
```

If either ***<nDatetime>*** is invalid, the function will return a -1. If the number of minutes remaining after all hours are accounted for exceeds 30, then an extra hour will be added to the return value.

Elap_mins()

This function takes two date/time values and returns the number of elapsed minutes between them. Its syntax is:

```
<nMins> := Elap_mins( <nDateTime1>,<nDateTime2> )
```

If either ***<nDatetime>*** is a invalid, the function will return a -1. If the number of seconds remaining after all minutes have been accounted for exceeds 30, then an extra minute will be added to the return value.

Listing 21.18 contains the user-defined functions for working with Date/Time variables. These functions call the Sec2hours() function from Listing 21.17, so be sure to link in that function, and compile with the **/n** option.

Listing 21.18 Date/Time Functions

```
static basedate := "01/01/80"
static half_day := 43200

function DateTime(dVar,cTime)
local nDatetime := -1, nSeconds := 0
local cDate, cSeconds
if dVar == nil .OR. cTime == nil
   return -1
```

```
else
   cDate     := alltrim(str(dVar - ctod(basedate),12,0))
   nSeconds  := val(cTime)*3600 + val(substr(cTime,4,2)) * 60 ;
             + val(substr(cTime,7,2))
   cSeconds  := alltrim(str(nSeconds))
   nDatetime := val(cDate + "." + cSeconds)
endif
return nDatetime

function GetDate(nDatetime)
local dDate := ctod("")
if nDatetime <> nil
   dDate := ctod(basedate) + int(nDatetime)
endif
return dDate

function GetTime(nDatetime)
local cTemp := str(nDatetime,15,5)
local nSecs := val(substr(cTemp,11,5))
cTemp := str(Sec2hours(nSecs),5,2)
return strtran(cTemp,".",":")+":00"

function Elap_days(nDt1,nDt2)
local nDays := -1, nDiff
if nDt1 == nil .or. nDt2 == nil
   return -1
else
   nDiff := abs( nDt1-nDt2 )
   nDays := int(nDiff)
   if nDiff - nDays >  half_day
      nDays++
   endif
endif
return nDays

function Elap_hours(nDt1,nDt2)
local nHours := -1,nDiff := 0,nRest := 0,nSecs := 0
if nDt1 == nil .or. nDt2 == nil
   return -1
```

```
else
   nDiff  := abs( nDt1-nDt2 )
   nHours := int(nDiff) * 24     // Number of days * 24 hours
   nSecs  := val(substr(str(nDiff,15,5),11,5))
   nHours += (nSecs/3600)
   if nSecs % 3600 > 1800
      nHours++
   endif
endif
return nHours

function Elap_mins(nDt1,nDt2)
local nMins := -1,nDiff := 0,nRest := 0,nSecs := 0
if nDt1 == nil .or. nDt2 == nil
   return -1
else
   nDiff  := abs( nDt1-nDt2 )
   nMins  := int(nDiff) * 1440    // Number of days expressed
                                  // in minutes
   nSecs  := val(substr(str(nDiff,15,5),11,5))
   nMins  += (nSecs/3600)
   if nSecs % 60 > 30
      nMins++
   endif
endif
return nMins
```

Summary

After completing this chapter your should feel comfortable writing simple and complex formulas in Clipper. You should know the various functions Clipper provides for mathematical operations and should be able to use several user-defined functions in your applications. You should also know how Clipper handles dates and times. If you need to work with times across dates, you should be able to work with the special date/time UDFs provided at the end of the chapter.

CHAPTER 22

Disks and Directories

A computer's disk drive can be viewed as a large filing cabinet. While information can be crammed into a filing cabinet with no sense of organization, it certainly loses its value when it is. The disk operating system (DOS) provides functions to organize the hard disk much as file folders can be used to organize the file cabinet. In this chapter, we will discuss how DOS organizes its disks and how Clipper can work within DOS's structure. We will also discuss the DOS environment and the functions Clipper provides to access that environment.

DOS environment

DOS is responsible for communicating between your application program and the computer hardware. Although other operating systems are available, DOS is the only platform that Clipper currently supports. Nantucket's future direction is to allow Clipper programs to be run on multiple platforms, such as OS/2 and Unix.

Clipper provides functions which access some of the information available from DOS. In addition, Clipper's powerful extend system allows an application to obtain status and request services from DOS using C and assembly language programs.

DOS status information

Clipper provides six functions to query DOS for various information. This information includes the operating system version, the system date and time, the current directory, the amount of space on the disk, and the environment string settings.

OS()—current operating system

The OS() function reports the current operating system and version that the application is running under. Its syntax is:

```
<cVersion>  := OS()
```

The **<cVersion>** returned will contain both the operating system name and the current version. For example, on a computer running under MS-DOS version 3.21, the OS() function would return "DOS 3.21" as a character string.

The OS() function can be useful to determine whether or not certain features in your program can be performed. For example, the DOS version must be 3.1 or above to run a Clipper network application. Here is the code for a user-defined function and an example that returns the version as a numeric value:

```
if DosVer() < 3.1
   ? "Cannot run networked application on this computer"
endif
return nil

function DosVer()
local cDos := os(), x, nVersion := 0.0
if (x := at(" ",cDos)) > 0
  nVersion := val(substr(cDos, x+1, 99))
endif
return nVersion
```

DATE()—current system date

The DATE() function returns a date variable containing the current system date. Its syntax is:

```
<dCurrent>  := DATE()
```

TIME()—current system time

The TIME() function returns a character string variable which contains the current system time. Its syntax is:

```
<cTime>  := TIME()
```

The **<cTime>** string is returned in the form of **HH:MM:SS** and is based on a 24-hour clock. The system time may also be returned as a number of seconds past midnight. The syntax for the SECONDS() function is:

```
<nSecond>   := SECONDS()
```

Dates and time are discussed in more detail in Chapter 21.

CURDIR()—current disk directory

The CURDIR() function returns the name of the current directory that the application is running in. Its syntax is:

```
<cDirectory>  := CURDIR(<cDrive>)
```

The **<cDirectory>** returned will be the directory name for the current specified drive. The directory name will not contain a leading or a trailing backslash (\) character.

<cDrive> is an optional parameter which specifies the drive letter to obtain the directory from. If not specified, the default will be the current drive. If the drive does not exist or it is not ready, a null string will be returned.

The CURDIR() function can be used to ensure that the program is running from the proper directory. Here's an example:

```
local where := curdir()
if where <> "APPL\PAYROLL"
   run cd\appl\payroll
endif
```

DISKSPACE()—free space on a disk drive

The DISKSPACE() function returns the number of bytes available on a disk drive. Its syntax is:

```
<nFreeBytes>  := DISKSPACE(<nDrive>)
```

<nFreeBytes> is an integer indicating the number of free bytes.

<nDrive> is an optional parameter which specifies the drive number to obtain the free space from. The drives are numbered starting at 1 for drive A:, 2 for B:, 3 for drive C:, and so on. If **<nDrive>** is not specified or is zero, the default will be the current drive.

The DISKSPACE() function can be used to ensure that the program has enough disk room to create needed files. The following listing checks to see if the current work area can be copied to drive A:.

```
local dbf_size
use CUSTOMER new
dbf_size := header() + (lastrec() * recsize())
if dbf_size > diskspace(1)    // Check drive A:
   ? "File too big to fit on diskette."
else
   copy to a:customer
endif
```

One caveat: The CURDIR() function and the DISKSPACE() function both accept a drive as a parameter; however CURDIR() expects the drive to be a character while DISKSPACE() expects a numeric parameter.

GETENV()—read DOS environment strings

DOS allows you to define environment strings using the SET command. Its syntax is:

```
SET <name> = <value>
```

One of the most common environment strings is the PATH setting. Clipper also makes use of DOS environment strings to control file handles, variable space, expanded memory, and so on. For example, the DOS command:

```
SET CLIPPER =E512;F35
```

instructs Clipper to restrict expanded memory usage to 512K bytes and to allow 35 file handles to be opened.

Clipper allows you to read the DOS environment strings using the GETENV() function. Its syntax is:

```
<cString>  := GETENV("cEnvironment_Variable>")
```

If the **<cEnvironment_Variable>** exists, its value will be returned in **<cString>**. If the DOS environment variable does not exist, a null string ("") will be returned. The variable name is not case-sensitive.

For example, in the listing below, the function returns the value of the **F** setting in the CLIPPER environment string. If no Clipper environment string has been specified, the function will return **20** which is the default maximum number of open files supported by Clipper.

```
function Get_F_Set()
local f_value := 20, clip_set := getenv("CLIPPER"), x
if !empty(clip_set)
   if (x := at("F", clip_set)) > 0
      f_value := val(substr(clip_set, x+1, 3))
   endif
endif
return f_value
```

Environment variables are most frequently used to pass directory and configuration information to programs. They are also very convenient ways of handling multiple configurations in a network environment.

Directories and files

When DOS formats a hard disk or a diskette, it sets aside room for a table to hold file names. This table will contain entries for any file found in the root area. In addition, DOS allows the user to create subdirectories, which allow the hard drive to be organized in an efficient manner. Each subdirectory also reserves room for a table to hold file names. These tables are called directories.

A directory is a list of all files either in the root directory or in a subdirectory. Clipper provides a function called DIRECTORY() to transfer this file list into an array. For compatibility reasons, Clipper also maintains the ADIR() function, although the stated preference is to use the newer function.

Directory entries

In order to effectively use the DIRECTORY() function, we must first explore how file names are stored within a DOS directory. Table 22.1 lists the information which is maintained for each file within a directory.

Table 22.1 File directory information

Name	*Contents*
Filename	Eight-character file name with three-character file extension. Normally, the file name and extension are separated by the period character.
File Size	The size of the file in bytes.
File Date	The date the file was last updated.
File Time	The time the file was last updated.

Attributes	A bit string indicating the type of file. It may be a combination of the following types. Normal file Hidden file System file Directory Read-only Volume label

Wildcards

The next item we need to be aware of is the wildcard. Wildcards are substitution patterns which allow functions to be performed on a select group of files. DOS uses two wildcard symbols, the asterisk and the question mark. The asterisk is used to match any files regardless of the length of the file name. The question mark is used to match any character in a particular position in the file name. For example, assume the files in Figure 22.1 exist in a directory:

Figure 22.1 Example files

```
SALES.RPT
SALESJT.RPT
SALESRA.RPT
SALESFRM.LPT
```

The pattern

*.RPT	Returns the first three files only.
*.?PT	Returns all four files.
SALES??.*	Returns the middle two files. The first file name is too short to match the pattern and the last file name is too long.

DIRECTORY()

The DIRECTORY() function allows you to select which files to place into the array by use of two parameters. The first parameter selects the pattern the file names must meet in order to be included. The second parameter specifies the attributes the file must have in order to be included. The syntax for the function is:

```
<array> := DIRECTORY( <pattern>,<attributes> )
```

<pattern> can be a single file or multiple files requested by using wildcard characters. If the **<pattern>** is not supplied, the default pattern will be *.*, which returns all files from the directory.

<attribute> can be any combination of the attribute codes listed in Table 22.2.

Table 22.2 Attribute codes

Code	*Meaning*
N	Normal files
H	Hidden files
S	System files
D	Subdirectories only
V	Volume label, excluding all other files.

The default is **Normal files**. Normal files are always included in the file list, unless the **Volume label** attribute is used.

For example, to fill the array with all normal files and all system files, the attribute would be set to "S". To get all normal files and directories, set the attribute to "D". The "V" attribute has special meaning since it excludes all files except the volume label. A function to return the volume label for any drive can be written as shown below:

```
function Vlabel( vdrive )
local test := directory( substr(vdrive,1,1) + ":\*.*", "V")
return if(empty(test),"No label found",test[1,1])
```

The function takes the drive letter specified and looks for the volume label using the directory function. If the directory function returns an empty array, then no volume label was found.

If the directory of a different subdirectory is needed, the subdirectory name can be included as part of the **<pattern>** specified to the DIRECTORY() function call.

The array that is returned is a multi-dimensional array with the structure shown in Table 22.3. The preprocessor constants are stored in DIRECTORY.CH.

Table 22.3 Directory array structure

Element #	*CONSTANT*	*Type*	*Contents*
1	F_NAME	Char	File name and extension.
2	F_SIZE	Numeric	Number of bytes in file.
3	F_DATE	Date	Date of last update.
4	F_TIME	Char	Time of last update.
5	F_ATT	Char	Attribute string

Here are a few examples of the DIRECTORY() function call.

```
#include "DIRECTRY.CH"
local  flist := directory("*.DBT")
? len(flist)                 // Displays number of .DBT files
aeval(flist,{|x| qout(x)})  // Display file names
flist := directory("CUSTOMER.DBT")
if !empty(flist)
   ?  flist[1,F_SIZE]        // Size of CUSTOMER.DBT file
endif
```

FILE()

The FILE() function is used to see if a file exists on the disk. Clipper will search the current directory and if the file is not found, will then search the Clipper path. The syntax for the function is:

```
<logical>  := FILE(<cFilespec>)
```

<cFilespec> can be any valid DOS file name or a wildcard pattern. If any file can be found that matches the file name, the function will return true. If after searching the current directory and the Clipper path, the file is not found, false will be returned.

The FILE() function will not search the DOS path, nor will it find hidden or system files. If you need to check for a hidden file, use the DIRECTORY() function.

The user-defined function in Listing 22.1 uses the Parse() UDF from Chapter 20 to search the path looking for a file. Its syntax is:

```
<logical>  := Inpath( <cFilespec> )
```

Listing 22.1 Inpath()

```
function Inpath(mfile)
local pathlist, jj, tfile
if ! file(mfile)
  pathlist := Parse( getenv("PATH"), ";" )
  for jj := 1 to len(pathlist)
     tfile := pathlist[jj] + "\" + trim(mfile)
     if file(tfile)
        return .T.
     endif
  next
else
  return .T.
endif
return .F.
```

Clipper's file searching

Clipper allows you to specify a search path that should be used when opening files. It also allows you to select the directory that Clipper should use as its default directory.

SET PATH

The SET PATH command allows you to create a list of file directories that Clipper should check if the requested file is not found in the current directory. Its syntax is:

```
SET PATH TO <cPath_list>
```

The **<cPath_list>** may be a literal constant or an expression surrounded by parentheses. The list of directories within the path can be separated by commas or semicolons. If a literal constant rather than a character expression is used, the SET PATH command line cannot be continued with the semicolon.

The path is only searched if the file is not found in the current directory and a path designation is not explicitly included in the file name. If the file is not found, Clipper will process the path list sequentially until it finds the file or the path is exhausted.

If no parameter is specified, the path list is released and Clipper will search only the current directory.

The listing below contains a user-defined function to set Clipper's internal path to the contents of the DOS path environment setting.

```
local dos_path := getenv("PATH")
if !empty(dos_path)
  set path to (dos_path)
endif
return nil
```

SET DEFAULT

When an application is started, DOS sets the current directory to the directory that the application is being run from. This is where Clipper will first look for its files. Clipper provides the SET DEFAULT command to instruct Clipper to look elsewhere for its files. Its syntax is:

```
SET DEFAULT TO <cPathspec>
```

The **<cPathspec>** may be a literal constant or an expression surrounded by parentheses. If the ***<cPathspec>*** includes a drive designation, it must be separated from the directory by a colon. If no parameter is specified the default directory will be the current DOS directory.

This command does NOT change the current directory, only where Clipper searches first for files. All temporary and low-level files will be put into the current DOS directory regardless of the setting of the SET DEFAULT command. The RUN command also accesses the current directory rather than the directory specified with SET DEFAULT.

Below is an example of a database being searched by company ID to determine the location of that company's files. Once the record is found, the SET DEFAULT value is set to the subdirectory specified in the DBF files.

```
function comp_def( comp_id )
use CONFIG.DBF new
locate all for CONFIG->id_code == comp_id
if found()
   set default to (CONFIG->directory)
endif
use
return nil
```

The low-level file functions are not affected by either the SET PATH or the SET DEFAULT settings. These functions are discussed in Chapter 24.

Running DOS programs

Clipper allows you to run a DOS application from within a compiled program. It also allows you to communicate with DOS through the ERRORLEVEL setting which DOS batch files can access. However, running an external DOS program requires memory, which is something that Clipper has an appetite for as well. For practical purposes, only small DOS utilities and commands, such as MD or FORMAT, should be run from within a Clipper application.

RUN

The RUN command executes a program or a DOS command. Its syntax is:

```
RUN <cCommand_or_program>
```

<cCommand_or_program> may either be a literal string or a character expression enclosed in parentheses. The command may be a DOS internal command, such as CLS or DIR, it may be an external DOS command, such as CHKDSK or PRINT, or it may be another application.

In order to shell out to DOS, Clipper executes another copy of COMMAND.COM and passes the requested command to it. This requires at least enough memory for COMMAND.COM (about 28K depending upon the DOS version) and for the application being run. If you expect to need memory for a DOS program, you should set the R Clipper environment string to reserve enough memory. This is done at the DOS prompt before loading the Clipper application. The DOS syntax is:

```
SET CLIPPER=R32
```

In addition to the memory requirements, Clipper must be able to find a copy of COMMAND.COM. Normally this file will reside in the root directory of the drive you booted your application from. If it is not there or the drive has been changed, a run-time error will occur. You can specify COMMAND.COM's location using the DOS environment string COMSPEC. The syntax is:

```
SET COMSPEC =<cDirectory name>
```

You can also use the RUN command to allow the user direct access to DOS. This is done by using the word COMMAND following the run command. When the user is done working in DOS, they should type in EXIT to return to the Clipper application. The DOS prompt can be changed to a message which reminds the user to type in EXIT when they are done in DOS.

You should never execute a memory-resident program using the RUN command. Doing so will almost guarantee memory problems on your return to the Clipper application.

Use RUN with extreme care. Its memory needs prevent it from being used for anything other than small commands and programs. There are several third-party libraries which take a different approach to RUNning DOS applications. If you frequently need to call DOS programs from within your Clipper application, you should investigate these libraries.

ERRORLEVEL()

The ERRORLEVEL() function is used to query and/or set the value of the DOS error level. This error level can be detected in a DOS batch file using the syntax:

```
IF ERRORLEVEL  <n>
```

By setting the error level, a Clipper program can direct a batch file's processing when the Clipper application terminates. The Clipper syntax for setting an error level is:

```
<nCurrent_setting> := ERRORLEVEL( <nNew_setting> )
```

The default error level at start-up is zero, indicating that no errors have occurred. If a Clipper application terminates with a fatal error, error level is set to one. If the program terminates normally, the error level is set to zero or the last value it was set to in the Clipper application.

Listing 22.2 illustrates a Clipper application setting the error level if the application does not complete processing. Presumably the batch file which called the Clipper application will respond by logging the problem somewhere. The batch file side of the application can best be understood by reading a DOS manual or user guide.

Listing 22.2 ERRORLEVEL() Example

```
begin sequence
   use CUSTOMER new
   if  ( header()+recsize()*lastrec() ) > diskspace(1)
      errorlevel(1)   // Couldn't fit CUSTOMER on backup disk
      break
   endif
   use VENDOR new
   if  ( header()+recsize()*lastrec() ) > diskspace(1)
      errorlevel(2)   // Couldn't fit VENDOR on backup disk
      break
   endif
   errorlevel(0)      // Everything ok, set ok errorlevel
end
quit
```

Summary

After reading this chapter you should know the functions that Clipper provides to access information from DOS. You should also know how to create a directory array and its structure. In addition, you should be able to establish paths and default directories for your Clipper commands. Finally, you should know how to execute DOS programs and commands from with a Clipper application.

Reexamination 90/005,727

The Original

Page 972 of Part IV

Is Missing

Initial : AR
Date : 6-8-2000

CHAPTER 23

Memo Files

The xBASE file structure supports two types of fields to handle character data. The character field is used for fixed length data. The memo field is used to handle variable length data. The actual memo contents are stored in a different file from the database. The database memo field contains a numeric pointer into the second file indicating where the text is written. The memo allows comments and notes to be written and associated with a particular record.

In this chapter we will discuss how memo fields are stored on the disk and the functionality Clipper provides for memo fields. We will also discuss how to correct some of the problems memo fields have, and present an alternative to using memo fields. Finally, we will present a text editing application to illustrate the power that is available in Clipper memo facilities.

Memo fields

A memo field sounds like a programmer's panacea: A method to store character data in a variable length record. Unfortunately, that is not quite the case. While the memo field provides variable length structure, its storage and update techniques in a .DBF file are very inefficient. This inefficiency is inherent in the .DBF file structure. While Clipper's implementation improves the problem a little bit, it would be difficult to correct the inefficiencies and maintain .DBF file compatibility.

How memos are stored in .DBF files

The technique used for storing memos on disk was inherited from the original implementation for dBASE files from Ashton-Tate.

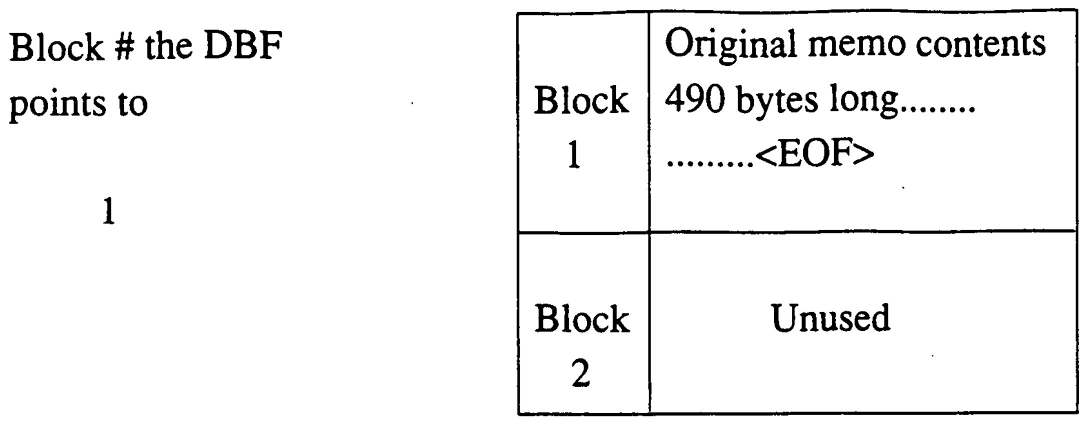
When Clipper was first introduced, it was described as a dBASE compiler. The language had to support all xBASE file formats. While the language has grown above and beyond the scope of the original dBASE language, the need to support dBASE files is still there.

When a database is created with a memo field in it, a file with the same name as the database, but with the .DBT extension, is also created. This file is where the text of the memos will be stored. The database itself has a 10-character numeric field, which holds the offset into the .DBT file.

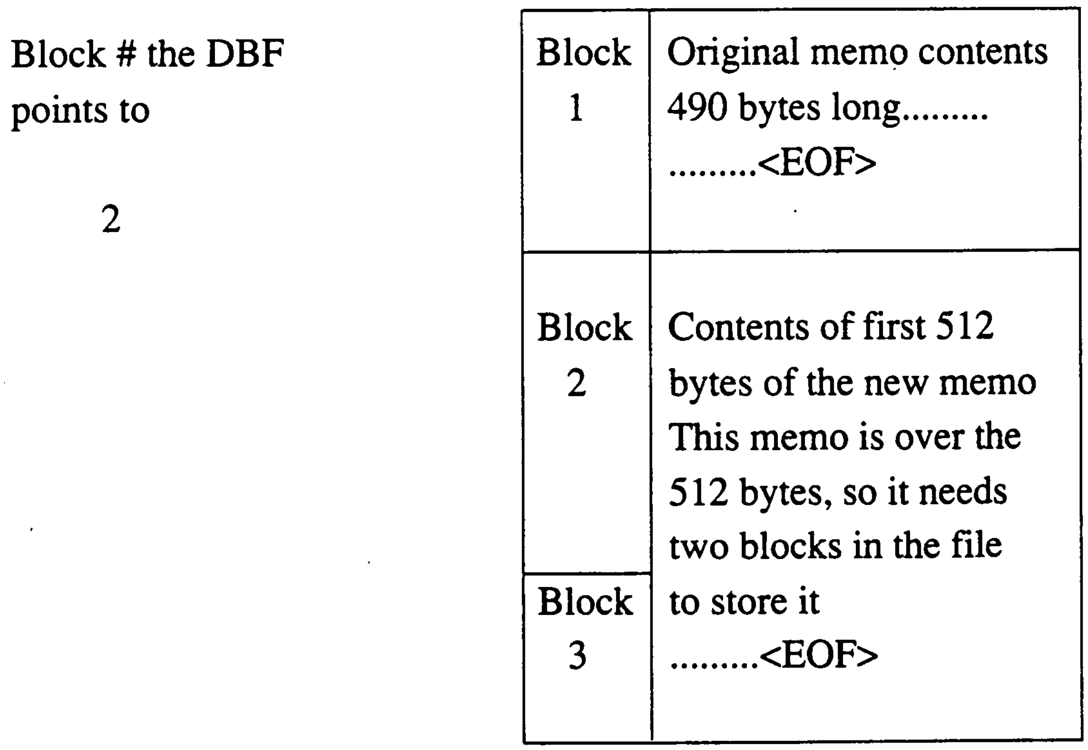
The .DBT file consists of a series of 512-byte blocks. Each block can contain only one memo. If the memo is small, then much of this 512-byte block is unused. When a memo field is requested, Clipper's database engine looks up the offset into the .DBT file, and starts reading from the offset until the end-of-file character is reached. (A memo file may have several end-of-file characters.) This value is returned as the memo field requested. When the memo is replaced into the database this mechanism works in reverse, using the offset and rewriting the memo into the .DBT file. (This is an improvement over dBASE, which appends edited memos to the end of the .DBT file instead of back where they were read from.)

If the above were a complete description of how the memo file was handled, then the growing .DBT file problem would not exist. Unfortunately, a problem occurs. If the new memo is larger than the original memo, and this additional size will cause a 512-byte boundary to be overwritten, then the edited memo will be appended to the end of the .DBT, using however many 512-byte blocks are needed. The problem occurs because the original block is now unused, and unavailable. Figure 23.1 indicates just how the problem occurs:

Figure 23.1 Memo growth


Block # the DBF points to
1
Block 1
Original memo contents 490 bytes long........ ....<EOF>
Block 2
Unused

After editing the memo, changing it to 520 bytes, and writing it back into the .DBT file, it looks like this:


Block # the DBF points to
2
Block 1
Original memo contents 490 bytes long......... ....<EOF>
Block 2
Block 3
Contents of first 512 bytes of the new memo This memo is over the 512 bytes, so it needs two blocks in the file to store it ....<EOF>

Notice that block one in the .DBT is now unused, and it never will be used. The only solution in Clipper is to copy the database to a new file, which creates a new .DBT file. The problem is now free to start all over again.

Why use memo fields?

If memo fields are inefficient in disk storage, why use them? The concept of the memo field can have some benefits. It allows the attachment of a note field to a particular record in the database. Memo fields can be used in many applications to provide the end-users with a method of writing notes or comments about the current record in a database file. Here are some examples:

An accounts receivable system might use a memo field for the collection agent to write notes about each customer contact. During the development of a new system, the users are given a pop-up utility to write notes and bugs to the programmer. These notes are stored in a memo field in a data file.

A legal time-billing system allows the lawyers to provide detailed descriptions for the clients as to the services they provided. The timeslip information is stored in a file with a memo field.

Character fields vs. memo fields?

Of course, since Clipper allows character fields in a database to be up to 1024 bytes long, you could use long character fields to store notes and comments. The decision between character and memo fields is based upon your application. Each character field will take up the maximum number of characters whether or not there is data in the field. Each memo field will take up ten bytes at a minimum. Records with no data in the memo field only need the ten bytes to save the pointer information. Fields with data will take up ten bytes plus the size of the memo field in 512-byte increments.

To determine which field type to use, examine the intent of your application. If your application expects only a few records to have data in them, the extra space used by character fields is wasted and a memo field would be worth considering. If you expect that most every record is going to contain information, then very little wasted space will be used in the file if you use character fields.

You also need to consider whether or not you expect frequent updates to the memos. If the memo data is constantly changing, you run the risk of having a lot of unusable space in the memo's .DBT file. If the file exceeds 16 megabytes, no more data can be put into the memo file. (16 megabytes is room for 32,000 512-byte blocks.) In this case, you would need to periodically reorganize the .DBT file through some utility operation in the application.

How memo variables are stored in memory

Once Clipper brings a memo field into memory it is treated just like any other character field. Clipper imposes a restriction that a memo field, or any string data, cannot be longer than 64K bytes. In most applications this maximum size is more than enough.

Working with memo field variables

Clipper provides a number of functions which are specifically intended for use with memo variables. These functions will work equally well on character variables. In addition, all string functions will work on memo variables (see Chapter 20).

Reading memo variables

There are two ways to read data into a memo variable. The method you use depends on where the memo's contents are stored. Clipper can read data into a memo variable from either a memo field in a database file or from a text file.

Memo field from database

If the database has a memo field, Clipper may access that memo field by an assignment operator. The syntax is:

```
<cMemo_variable> := <cAlias> -> <cMemofield>
```

If the ***<cAlias>*** is not supplied, the memo field is assumed to be in the current work area. You can also use the FIELDGET() function to assign the memo field to a memory variable. Its syntax is:

```
<cMemo_variable> := FIELDGET( <nField_no> )
```

Clipper's database drivers automatically translate the request for a memo field into a block reference to be read from the .DBT file.

For example, Listing 23.1 shows a function to search all memo fields in a .DBF file for a particular character string. The function returns an array of record numbers which contain memos where the text was found. The syntax is:

```
<array>  := Find_memos(<cString>,<nField_no>)
```

Listing 23.1 Find_memos()

```
function Find_memos(cString,nField)
local arr_ := {}
go top
do while !eof()
   if cString $ fieldget(nField)
      aadd(arr_,recno())
   endif
   skip +1
enddo
return arr_
```

MEMOREAD()

The MEMOREAD() function is used to read a file from the disk into a memo variable. Its syntax is:

```
<cMemo>  :=  MEMOREAD( <cFilename> )
```

If *<cFilename>* is not in the current directory, Clipper will search the DOS path for the file. It will not search the path specified by Clipper's SET DEFAULT TO command nor by Clipper's SET PATH command. If the file cannot be found in either the current directory or the DOS path, an empty string ("") will be returned.

The maximum file size which can be read by MEMOREAD() is 65,519 (64K) bytes.

If **<cFilename>** is on a network, Clipper will attempt to open the file in a shared, read-only mode. If the file cannot be opened in that mode, Clipper returns an empty string ("").

The function in Listing 23.2 shows how a text file can be read into a variable and then edited by MEMOEDIT().

Listing 23.2 MEMOREAD()

```
function Miniedit( cFile )
local mText := memoread( cFile )
return memowrit( cFile,memoedit(mText) )
```

Editing memo fields

Clipper provides a tool for complete edit capabilities of memo and large character string variables. The MEMOEDIT() tool provides text-editing capabilities in a single function. In addition, the function allows the programmer to gain control during the middle of editing.

MEMOEDIT()

The MEMOEDIT() function is used to edit a memo variable. It can also be used to display the contents of a memo variable. Its syntax is:

```
<mString> := memoedit(<mString>, ;
<nTop_row>,<nTop_column>,<nBottom_row>, <nBottom_col>, ;
<lEdit?>,<cmemo_udf>, <nLinelength>,<nTabsize>, ;
<nTextRow>,<nTextCol>,<nWindowRow>,<WindowCol> )
```

The ***<mString>*** is the string to be edited. This string gets transferred into the memory buffer that MEMOEDIT() works on. The function will return either the original string (if the user presses the Esc key) or the modified string if the user presses the Ctrl-W combination. Depending upon the size of the string, MEMOEDIT() can be quite memory hungry, since it must hold a copy of the original string as well a copy of the edited string. If you choose to run MEMOEDIT() on a 16K character string, the function will require at least 32K of memory.

You should add a call to the MEMORY() function to determine if sufficient memory is present to run MEMOEDIT(). For example, the Mem_chk() function in Listing 23.3 returns true if there is more than three times the amount of memory than the length of the memo to be edited. Three times is an estimated factor to allow two copies of the string and some growth room in the memo buffer.

Listing 23.3 Mem_chk()

```
function Mem_chk(param)
param := if(valtype(param)=="C",len(param),param)
// Check largest contiguous block of storage //
return ( param * 3 < memory(1)*1024 )
```

The ***<nTop_row>***,***<nTop_column>***,***<nBottom_row>*** , and ***<nBottom_col>*** parameters represent the coordinates in which the memo is edited. The rows can range from zero to MAXROW() and the columns for zero to MAXCOL(). If the coordinates are not supplied, the entire screen will be used. The coordinates are not framed by the MEMOEDIT() function. If you wish to create a box around the MEMOEDIT() window, you can use the @ BOX command or the @...TO syntax.

<lEdit?> is a logical value which indicates if the memo may be updated or is simply being displayed. If the value is true, the memo may be edited. A false value causes the memo to be displayed and allows the user to browse through the memo. The default is true.

<cMemo_udf> is an optional user-defined function name. If this parameter is specified, control is passed to this UDF during the editing of the memo. This allows your program to provide extra features and options during MEMOEDIT(). The example program at the end of this chapter includes a MEMOEDIT() UDF.

You can also specify **.f.** instead of a UDF. If false is specified, the memo is displayed and MEMOEDIT() immediately terminates. For example, Listing 23.4 displays a memo on the screen and waits for the user to press any key.

Listing 23.4 MEMOEDIT() display only

```
function Memodisp(cMemo)
local oldscrn := savescreen(10, 10, 20, 60)
@ 10,10 to 20,60 double                    // Draw a box
memoedit( cMemo,11,11,19,59,.f.,.f. )  // Display the memo
inkey(500)                                 // Wait for any key
restscreen(10, 10, 20, 60, oldscrn)
return nil
```

<nLinelength> indicates the length of lines within the text of the memo. If a line of text is longer than this length, it will be word-wrapped to the next line. If the width of the window is smaller than the line length, horizontal scrolling will work. If you do not specify the line length, Clipper will subtract the left column coordinate from the right, and use this result as the length.

The ***<nTabsize>*** determines how many spaces tab characters should be converted to. The default is four spaces. When the user presses the tab key, this number of spaces will be inserted into the memo. If the memo contains a tab character while being displayed, Clipper will convert that tab character to the appropriate number of spaces.

<nTextRow> and ***<nTextCol>*** determine where the cursor should be positioned initially within the memo. The default location is the first memo character (i.e. row one, column zero).

<nWindowRow> and ***<nWindowCol>*** determine where the cursor should be positioned initially within the window. The default location is row zero and the current cursor column.

The sample application at the end of this chapter employs several techniques in the MEMOEDIT() user-defined function to perform macro substitution and search and replace operations.

Memoedit user-defined function

The MEMOEDIT() UDF allows the programmer to control processing while the user is in MEMOEDIT(). This UDF is a black-box process to the function. It merely passes it three parameters and expects a single parameter to be returned. As long as your function returns what Clipper expects, the internals of your function are invisible to MEMOEDIT().

To write a MEMOEDIT() UDF, we need first to understand when Clipper will call the function and what information it will pass to it. Clipper calls the function at three times. These times are designated by a mode parameter that Clipper sends the UDF. In addition to the mode parameter, Clipper also sends the current line and column of the memo.

MODE 3 - Initialization Mode. Before MEMOEDIT() starts processing, a call will be made to the user-defined function. This call is made to allow your function to specify the formatting modes. The formatting modes are listed in Table 23.1. The value your function returns instructs MEMOEDIT() what to do next. MEMOEDIT() will continue to call your function in initialization mode until your function returns a zero. The example code fragment in Listing 23.5 shows a portion of the memo UDF to toggle word wrap OFF and insert mode ON before instructing MEMOEDIT() to leave initialization mode.

Table 23.1 Formatting modes

Mode	*Default*	*Return value to toggle mode*	
word-wrap	ON	34	ME_TOGGLEWRAP
scroll	ON	35	ME_TOGGLESCROLL
insert	OFF	22	K_INS (in INKEY.CH)

Listing 23.5 Sample UDF

```
#include "MEMOEDIT.CH"
#include "INKEY.CH"
function ME_UDF( me_mode, me_row, me_col )
static me_loop :=0
if me_mode == ME_INIT      // MEMOEDIT.CH  (Mode 3)
   me_loop++
      do case
      case me_loop == 1
         return ME_TOGGLEWRAP
      case me_loop == 2
         return K_INS
      otherwise
         return 0
      endcase
   endif
```

MODE 0 - Idle Mode. If MEMOEDIT() has no pending keystrokes to process, it will call your function with an idle mode. This mode is most frequently used to update the screen display of row and column. You could also use this mode to display the time on the screen, or any other background processing that needs to be performed. The example in Listing 23.6 shows a sample UDF to update the screen display during idle mode.

Listing 23.6 Idle mode example

```
#include "MEMOEDIT.CH"
#include "INKEY.CH"
function ME_UDF( me_mode, me_row, me_col )
if me_mode == ME_IDLE      // MEMOEDIT.CH  (Mode 0)
   @ 2,50 say me_row   pict "9999"
   @ 2,60 say me_col+1 pict "9999"  // Columns start at 0
   @ 2,70 say time()
endif
```

MODES 1/2 keystroke exception modes. When MEMOEDIT() detects a keystroke that it does not handle, it hands control over to your user-defined function. This call is made to allow your function to specify what you want MEMOEDIT() to do. You do this by returning a numeric value instructing MEMOEDIT() what to do. The possible return values that MEMOEDIT() can be sent are listed in Table 23.2.

Table 23.2 MEMOEDIT() UDF return values

VALUE	*MEMOEDIT.CH or INKEY.CH*	*Action to perform*
0	ME_DEFAULT	Perform default action for key
1	K_CTRL_A	Move left one word
2	K_CTRL_B	Reformat the memo's contents
3	K_PGDN	Move to next block in memo
4	K_RIGHT	Move right one character
5	K_UP	Move up one line
6	K_END	Move right one word
7	K_DEL	Delete character at cursor
8	K_BS	Delete character to left of cursor
9	K_TAB	Insert TAB or spaces
13	K_ENTER	Move to beginning of next line
18	K_PGUP	Move to previous block
19	K_LEFT	Move left one character
20	K_CTRL_T	Delete word to right
22	K_INS	Toggle insert on or off
23	K_CTRL_END	Finish editing and save memo
24	K_DOWN	Move down one line
25	K_CTRL_Y	Delete the current line
27	K_ESC	Abort edit and return original
29	K_CTRL_HOME	Move to beginning of window
30	K_CTRL_PGDN	Move to bottom of memo
31	K_CTRL_PGUP	Move to beginning of memo
32	ME_IGNORE	Ignore the keystroke
33	ME_DATA	Treat the keystroke as data
34	ME_TOGGLEWRAP	Toggle word-wrap mode
35	ME_TOGGLESCROLL	Toggle scroll mode
100	ME_WORDRIGHT	Move right one word
101	ME_BOTTOMRIGHT	Move to bottom right

When the user function is called, mode 1 will be passed if the memo has not been altered, mode 2 is passed once the memo is changed. This allows you to display a warning if the user wishes to abort an updated memo without saving it.

To handle the keystroke exceptions, you would first capture the value from the LASTKEY() function. This value would then be tested in a DO CASE sequence to determine what return value should be sent back to MEMOEDIT(). In addition to returning a value to MEMOEDIT(), your function may contain additional code as well. Listing 23.7 illustrates a code fragment which will toggle insert mode and also change the cursor shape if the user presses the INSERT key during the editing of the memo. The example program at the end of this chapter also illustrates some keystrokes that can be handled in the MEMOEDIT() user-defined function.

Listing 23.7 Keystroke exception UDF

```
#include "MEMOEDIT.CH"
#include "INKEY.CH"
#include "SETCURS.CH"

function ME_UDF( me_mode,me_row,me_col )
local keypress
if me_mode == ME_UNKEY .or. ;    // MEMOEDIT.CH (Mode 1)
   me_mode == ME_UNKEYX          // (Mode 2)
   keypress := lastkey()         // Determine key pressed
   do case
   case keypress == K_INS        // Insert key
      if readinsert()            // Is insert currently on?
         setcursor(SC_NORMAL)    // Underline cursor
      else
         setcursor(SC_SPECIAL1) // Full block cursor
      endif
      return K_INS               // Toggle cursor ON/OFF
   case keypress == K_F5         // F5 - insert date
      keyboard dtoc(date())      // Insert system date
      return ME_DEFAULT
   otherwise
```

```
      return ME_DEFAULT       // Zero - default action
   endcase
endif
return ME_DEFAULT
```

Displaying/printing memo fields

While MEMOEDIT() can be used to display memo variables, there are times when more control over the display is needed. A good example is printing. Since printing occurs a line at a time, MEMOEDIT() cannot be used to print individual lines from the memo. The MLCOUNT(), MLPOS(), and MEMOLINE() functions provide this degree of control.

MLCOUNT()

MLCOUNT() returns an integer indicating how many lines are found in the memo field. Its syntax is:

```
<nCount> := MLCOUNT(<mString>,<nWide>,<nTabsize>,<lWrap>)
```

<nCount> will receive the total number of lines in the memo. The contents of **<nWide>**, **<nTabsize>**, and **<lWrap>** are all taken into consideration in determining the number of lines the memo will take to display.

<mString> is the memo variable to be analyzed. It may also be a character string rather than a memo field.

<nWide> is the width of each line. It can be in the range of 4 to 254. If not specified, the default value is 79 characters wide.

<nTabsize> indicates how many spaces tabs should be expanded to. The tab size defaults to four if not specified. If the tab size is larger than or equal to the **<nWide>** value, it will automatically be converted to **<nWide>** minus one.

<lWrap> is a logical setting, where **.t.** indicates that the lines should be word-wrapped and **.f.** indicates they should not. If not specified, the default is true.

MEMOLINE()

Memoline extracts a line from a memo variable. Its syntax is:

```
<cVariable> :=MEMOLINE(<cMemo>,<nLength>,;
            <nLine>,<nTab>,<lWrap>)
```

<cVariable> will contain the extracted line from the memo variable. If the line contains fewer characters than specified by **<nLength>**, it will be padded with spaces to fill the line. If the **<nLine>** is greater than the number of lines in the memo, an empty string ("") will be returned.

<cMemo> is the memo variable to be analyzed. It may also be a character string rather than a memo field.

<nLength> is the width of each line. It can be in the range of 4 to 254. If not specified, the default value is 79 characters wide.

<nLine> indicates which line to extract from the memo. If not specified, the default will be line one. If the value of **<nLine>** is larger than the number of lines in the memo, the function will return a null string.

<nTab> indicates how many spaces tabs should be expanded to. The tab size defaults to four if not specified. If the tab size is larger than or equal to the **<nWide>** value, it will automatically be converted to **<nWide>** minus one.

<lWrap> is a logical setting, where **.t.** indicates that the lines should be word-wrapped and **.f.** indicates they should not. If not specified, the default is true.

MEMOLINE() and MLCOUNT() are frequently used together to display a memo's contents to the printer. For example, Listing 23.8 shows a sample program to print a form letter from a memo field and the customer database file.

Listing 23.8 Form letter

```
STATIC nWide := 60

function Form_Lett( mText )
local num_lines := mlcount(mText,nWide), jj
use CUSTOMER new
go top
do while !eof()
  ? CUSTOMER->Name
  ? CUSTOMER->Address
  ? trim(CUSTOMER->City)+", "+CUSTOMER->State
  ?? " "+CUSTOMER->Zip
  ?
  for jj := 1 to num_lines
     ? memoline(mText,nWide,jj)
  next
  ?? chr(12)          // eject page
  skip +1
enddo
return nil
```

MLPOS()

The MLPOS() function is used to determine the character position that a line of a memo starts at. Its syntax is:

```
<nPosition> :=mlpos(<cMemo>,<nLength>,;
             <nLine>,<nTab>,<lWrap>)
```

<nPosition> will contain an integer value indicating where the requested line is offset into the string. If the **<nLine>** value is larger than the number of lines in the string, MLPOS() will return the length of the string.

<cMemo> is the memo variable to be analyzed. It may also be a character string rather than a memo field.

<nLength> is the width of each line. It can be in the range of 4 to 254. **<nLength>** is a required parameter.

The ***<nLine>*** indicates which line to look for. The **<nLine>** parameter is required. If the value of **<nLine>** is larger than the number of lines in the memo, the function will return the length of the string.

<nTab> indicates how many spaces tabs should be expanded to. The tab size defaults to four if not specified. If the tab size is larger than or equal to the **<nWide>** value, it will automatically be converted to **<nWide>** minus one.

<lWrap> is a logical setting where **.t.** indicates that the lines should be word-wrapped and **.f.** indicates they should not. If not specified, the default is true.

MLPOS() could be used to extract a section from a memo. For example, Listing 23.9 finds the fifth line within a memo and creates a new string starting at that line.

Listing 23.9 MLPOS() example

```
nStart     := mlpos(cMemo,79,1,4,.t.)
cNew_memo  := substr(cMemo,nStart,9999)
```

Positioning memo fields

Clipper 5 provides functions to determine the byte position of a row/column coordinate within a memo field or the row/column coordinates a given byte would be displayed at.

MLCTOPOS()

The MLCTOPOS() function takes a memo field and formatting information and returns an offset byte into the memo field. The offset is the byte position that corresponds to a particular row and column coordinate. Its syntax is:

```
nByte := MLCTOPOS(<cText>, <nWide>, <nLine>,;
        <nCol>, <nTab>, <lWrap>)
```

<nByte> is an integer value representing the offset position in the memo string.

<cText> is the actual memo string to be examined.

<nWide> is the memo display width.

<nLine> is the row number and ***<nCol>*** is the column position to look for. The row number starts at one, the column number is based from zero.

<nTab> is the number of spaces between tab stops. If not supplied, it defaults to four.

<lWrap> is a logical value indicating whether or not word-wrapping should be performed. If true, which is the default, word-wrapping is considered on.

For example, assume your help system uses MEMOEDIT() to display help text on the screen. If the user moves to a word within the help text and presses the F6 key, the HELP system will extract the word at the cursor and look for HELP text about the selected word. MEMOEDIT() will handle the cursor position and pass to the MEMOEDIT() user-defined function the current screen row and column. The MLCTOPOS() function is used to take these coordinates and extract the appropriate word from the HELP text. Listing 23.10 illustrates an example.

Listing 23.10 MLCTOPOS() example

```
#include "MEMOEDIT.CH"
#include "INKEY.CH"

function ME_UDF( me_mode,me_row,me_col )
local keypress, nByte, cWord, cScr
field cHelp
if me_mode == ME_UNKEY .or. ;  // MEMOEDIT.CH (Mode 1)
```

```
    me_mode == ME_UNKEYX          // (Mode 2)
    keypress := lastkey()         // Determine key pressed
    do case
    case keypress == K_F6         // F6 - Hyper lookup
       nByte := mlctopos(cHelp,60,me_row,me_col)
       cWord := substr(cHelp,nByte,12)
       select HELP
       seek upper(cWord)
       if found()
          *
          * Display a second box to describe the word found
          *
          cScr :=savescreen(8,20,15,60)
          @ 8,20 to 15,60 double
          @ 8,21 say " "+trim(cWord)+" "
          memoedit( HELP->Text,9,21,14,59,.f.,,.f.)
          inkey(500)
          restscreen(8,20,15,60,cScr)
          *
       endif
       return ME_DEFAULT
    otherwise
       return ME_DEFAULT          // Zero - default action
    endcase
endif
return ME_DEFAULT
```

MPOSTOLC()

The MPOSTOLC() function takes a memo field and formatting information and returns an array containing the row and column that a particular offset byte would be displayed at. Its syntax is:

```
<aBytes> := MPOSTOLC(<cText>, <nWide>, <nPos>, <nTab>, <lWrap>)
```

<aBytes> is a two-element array of integer values representing the row and column positions that the specified byte position would be displayed at.

<cText> is the actual memo string to be examined.

<nWide> is the memo display width.

<nPos> is the byte position within the memo string to be displayed. It starts at one, i.e. the first character of the string is offset one, not zero.

<nTab> is the number of spaces between tab stops. If not supplied, it defaults to four.

<lWrap> is a logical value indicating whether or not word-wrapping should be performed. If it's true, which is the default, word-wrapping is considered ON.

The MPOSTOLC() function is the inverse to the MLCTOPOS() function.

For example, Listing 23.11 will search all memo fields on a disk for a requested string. If the string is found, the memo will be brought up for editing, with the cursor positioned on the found word.

Listing 23.11 MPOSTOLC() example

```
function FindMemo(cWord)
local found_one := .f.,arr_ := {}, x := 0
select BOOKS
go top
do while !eof()
   if !empty(BOOKS->Desc)        // Memo field
      if (x := at(cWord,BOOKS->Desc)) > 0
         arr_ := mpostolc( BOOKS->Desc,44,x,4)
         found_one := .t.
         exit
      endif
   endif
   skip +1
enddo
if found_one
```

```
      replace BOOKS->Desc with  ;
                memoedit(BOOKS->Desc,8,20,19,65,.t.,,44,4,;
                         arr_[1],arr_[2])
   endif
   return nil
```

Saving memos to disk

The contents of a memo variable may be saved to either a memo field in a .DBF file or a text file in DOS.

REPLACE Command

If the database has a memo field, Clipper uses the REPLACE command to update that memo field. The syntax is:

```
REPLACE <cAlias> -> <cMemofield>  WITH <cMemo_string>
```

If ***<cAlias>*** is not supplied, the memo field is assumed to be in the current work area. You can also use the FIELDPUT() function to write the memo variable into the DBF file. Its syntax is:

```
FIELDPUT( <nField_no>,<cMemo> )
```

Clipper's database drivers automatically translate the request to write the memo field into a block reference to be written to the .DBT file.

MEMOWRIT()

The MEMOWRIT() function takes a memo variable and creates a text file with its contents. Its syntax is:

```
<logical> := MEMOWRIT(<cFile>,<cMemo_variable>)
```

If ***<cFile>*** is successfully created, a logical true value is returned. If not, false will be returned. The created file will contain soft carriage returns if any were present in the memo variable. The MEMOTRAN() function described in the next section can be used to convert the soft carriage returns to hard carriage returns.

Memo file maintenance

Memo fields are, unfortunately, not self-maintaining structures. If your system makes use of memo fields, your end-users will occasionally need to optimize the memo and reclaim unused space. If you are converting an application to Clipper 5 from earlier versions, you'll need to fix the memo files before Clipper 5 can update the field. The steps for accomplishing this uplate are described in this chapter. See section entitled "Converting Summer '87 memos to Clipper 5".

Packing the memo file

When a .DBF file is packed, all deleted records are removed and the file is rewritten to the disk. This process reduces the physical storage space the .DBF file uses. Unfortunately, it does not reorganize the corresponding .DBT. To reclaim unusable space in a memo file, you need to create a copy of the .DBF. This new copy will contain a packed .DBT file.

The procedure to create a new file, hence packing the DBT is:

```
use MEMOS new
copy to NEW_MEMO
close databases
rename MEMOS.DBF to MEMOS.BAK     // Back up the old DBF
rename MEMOS.DBT to MEMOS.TBK     //    and DBT files
rename NEW_MEMO.DBF TO MEMOS.DBF
rename NEW_MEMO.DBT TO MEMOS.DBT
```

If your application program uses memos, be sure to include a menu option to allow the user to "pack" their memo files.

Changing carriage returns

Clipper uses the ASCII code of 141 to designate a soft carriage return in a memo field. A soft carriage return is a carriage return created when Clipper cannot fit the word on the current line and wraps it to the next line. A hard carriage return occurs when the user actually presses the enter key. The ASCII code for a hard carriage return is the number 13.

Some applications, notably Clipper's own REPORT FORM command, do not recognize carriage returns. In REPORT FORM, a semi-colon is used to designate text that should appear on the next line. In order to work with other applications and report form, Clipper provides the MEMOTRAN() function to convert carriage returns. Its syntax is:

```
<cString> := MEMOTRAN( <cString>,<cRepl_hard>,<cRepl_soft> )
```

<cString> is the memo string that should be changed. This is a required parameter; it can be a character string as well as a memo field.

<cRepl_hard> is the character that all occurrences of hard carriage returns should be replaced with. If not specified, the default is a semi-colon, which allows REPORT FORM to work with the memo's contents.

<cRepl_soft> is the character that all occurrences of soft carriage returns should be replaced with. If not specified, the default is a space character.

For example, this function call replaces all soft carriage returns with a semicolon. By changing the soft return to a semicolon, various dBASE dialects could read this string and insert a new line at each occurrence of the semicolon.

```
memowrit( "NEWFILE.TXT", memotran(cMemo,,";") )
```

Converting Summer '87 Memos To Clipper 5

In Clipper, word-wrapping occurs when a line of text is too wide to fit within the line width specified in the call to MEMOEDIT(). The line is broken into two lines by inserting a soft carriage return/line feed after the rightmost word that will fit within the specified line width.

In earlier versions of Clipper, when a soft carriage return is inserted, a single space character is removed from the text at the point where the soft carriage return is inserted. If the text is later reformatted using a different line width, each soft carriage return is replaced by a single space. This generally results in equivalent text, regardless of the formatting width. A problem arises, however, if a single word is too long to fit within the specified line width. In this case, a soft carriage return is inserted in the middle of the word. If the text is later reformatted with a greater line width, the word will appear as two words because the soft carriage return is replaced with a space.

In Clipper 5, the insertion of soft carriage returns has been enhanced so that changes in the content of the text are never permitted. When a soft carriage return is inserted between two words, the space characters between the two words are preserved. When text is reformatted, any soft carriage returns are simply removed. This leaves the text in its original form and properly handles the case where a soft carriage return has been inserted in the middle of a word.

If a text string formatted using Summer '87 MEMOEDIT() is reformatted using any of the Clipper 5 text handling functions, words that were separated by a soft carriage return will be run together because the soft carriage return is not replaced with a space.

You can use the STRTRAN() function to adjust memo fields from earlier versions of Clipper. The syntax is:

```
STRTRAN( <text>, chr(141)+chr(10), " " )
```

To convert memo values in an existing .DBF file, the following two-line program can be used:

```
use <cDatabase>
replace all <idMemo> with ;
      strtran(<idMemo>,chr(141)+chr(10), " ")
```

The memo translation process should be performed before any text in the .DBF file is reformatted and written using the Clipper 5 MEMOEDIT() function. As with any operation of this kind, be sure to back up your databases before you run the conversion program.

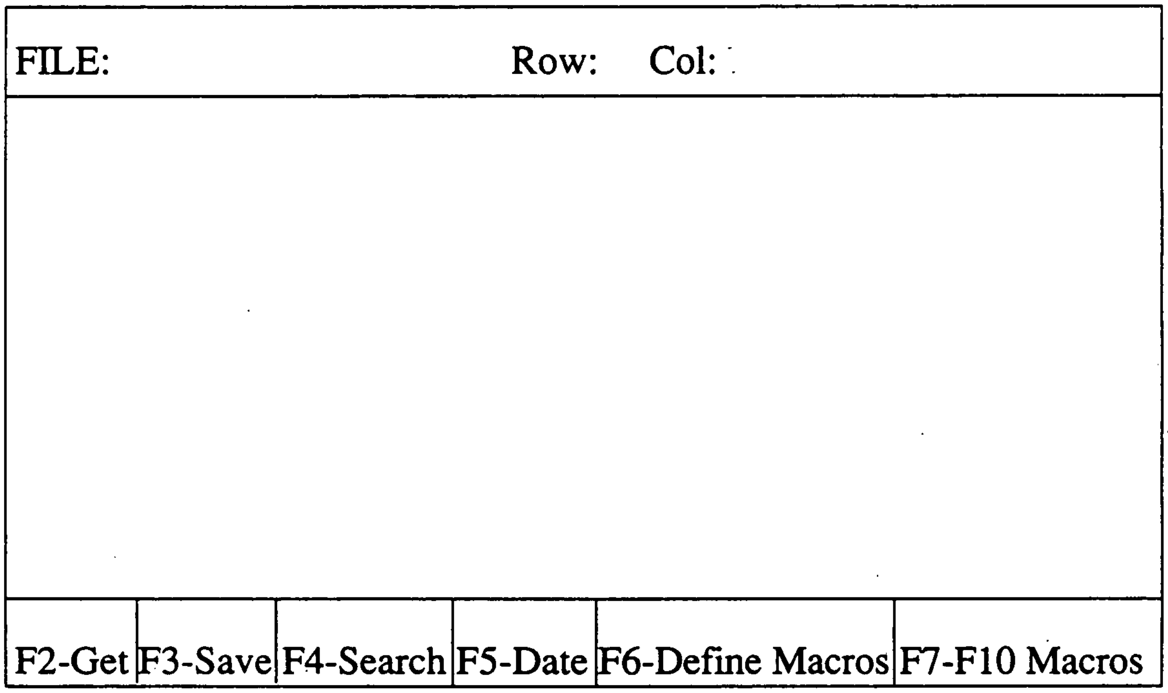
Sample memo application programs

To illustrate the power of memo handling, our example program will contain a complete text editor. The editor will be able to select a disk file to read, edit the contents, and write the file back to disk. During the edit mode, you will be able to search and replace strings, insert the system date, and create macros.

The code to the text editor is in Listing 23.12. We've commented it to illustrate points we've covered in the chapter. While this editor is not going to replace WordPerfect or Microsoft Word, it is a handy tool which can be included within your Clipper application. It could easily be customized to allow the end-user to store the memos in a .DBF file and search the .DBF for which memo to edit.

The main screen for our editor is illustrated in Figure 23.2.

Figure 23.2 Editor main screen



The function keys listed in Table 23.3 are used to control the editing process.

Table 23.3 Edit function keys

F2	Get a file from the disk
F3	Save the text to the disk
F4	Search and replace strings
F5	Insert current date
F6	Define macro keys F6 through F10
F7	Macro number one
F8	Macro number two
F9	Macro number three
F10	Macro number four

Listing 23.12 Text editor

```
// Pre-processor settings //

#define DEF_SIZE 2048
#define LINE_LNG 255
#include "MEMOEDIT.CH"
#include "INKEY.CH"
#include "BOX.CH"
#include "SETCURS.CH"
MEMVAR getlist

function Cdg_Edit
local mx := Maxrow(),mc := Maxcol()
local backscr := savescreen(0,0,mx,mc)
local msearch := space(8),mreplace := space(8)
local mcount := 1,msave

private keepgoing := .t.,memo_buff := space(DEF_SIZE)
private me_file := space(12),me_command := 0
private me_macros := {space(15),space(15),space(15),space(15)}
```

```
//  Set the colors and draw the screen  //
setcolor('W+/B')
cls
dispbox(01, 00, mx, mc, 2)
@ 02,03 say "FILE:"
@ 02,60 say "Row:      Col:"
@ 03,01 say replicate( chr(196),mc-1)
@ mx-2,01 say " ────────┬─────────┬────────────┬──────── "+ ;
              "┌──────────────────────────────────── "
@ mx-1,01 say "  F2-Get | F3-Save | F4-Replace | F5-Date "+;
              "| F6-Define Macros | F7-F10 Macros  "
@ mx,  01 say "╘════════╧═════════╧════════════╧════════ "+ ;
              "╘══════════════════╧════════════════ "
setcolor ('W/B')

do while keepgoing
   memo_buff :=memoedit(memo_buff, 4, 1, mx-3,;
               mc-1, .t., "ME_UDF", LINE_LNG)
   if !empty(me_command)
      do case
      case me_command == 1
         keepgoing := .f.
         loop
      case me_command == 2        // Get a new file
         memo_buff := memoread(me_file)
      case me_command == 3        // Save existing file
         memowrit(me_file,memo_buff)
      case me_command == 4        // Search/replace a string
         msave := savescreen(2,24,2,79)
         @ 2,60 say space(18)
         @ 2,24 say "SEARCH:"  get msearch
         @ 2,42 say "REPLACE:" get mreplace
         @ 2,62 say "COUNT:"   get mcount   ;
                pict "99999" valid mcount>=0
         read
         if lastkey() <> K_ESC
            mcount := if(mcount=0,99999,mcount)
            memo_buff := strtran(memo_buff,trim(msearch),;
                              trim(mreplace),1,mcount)
         endif
         restscreen(2,24,2,79,msave)
      endcase
      me_command := 0
   endif
enddo
restscreen(0,0,mx,mc,backscr)
return nil

function ME_UDF(me_mode,me_row,me_col)
local me_key := lastkey(),me_save
MEMVAR me_macros,me_file,me_command

if me_mode == ME_INIT       // Initialization mode
   return ME_DEFAULT        // Tell memoedit to start
elseif me_mode == ME_IDLE   // Idle mode, update row/column
   @ 02,10 say me_file
   @ 02,65 say me_row    pict "9999"
   @ 02,74 say me_col+1  pict "9999"  // Starts at zero
else
   do case
   case me_key == K_INS          // Insert key
      if readinsert()            // Is insert on?
         setcursor(SC_NORMAL)    // Underline cursor
      else
         setcursor(SC_SPECIAL1)  // Full block cursor
      endif
      return K_INS      // Toggle cursor ON/OFF
   case me_key == K_ESC .or. me_key == K_CTRL_END
      me_command := 1
      return me_key
   case me_key == K_F2
      @ 2,10 get me_file pict "!!!!!!!!!!!!"  ;
            valid file(me_file) .or. lastkey() == K_UP
      read
      if lastkey() <> K_ESC
         me_command :=2
         return K_ESC
      endif
   case me_key == K_F3         // Get a file name to save
      @ 2,10 get me_file pict "!!!!!!!!!!!!" ;
```

```
                valid !empty(me_file)
         read
         if lastkey() <> K_ESC
            me_command := 3    // Set command to save contents
            return K_CTRL_END
         endif
      case me_key == K_F4
         me_command :=4       // Set command to search/replace
         return K_CTRL_END      // Save the memo's contents
      case me_key == K_F5
         keyboard dtoc(date()) // Insert a date into the memo
      case me_key == K_F6
         me_save := savescreen(9,29,14,53)
         @ 9,29,14,52 box B_DOUBLE_SINGLE + " "
         @ 10,30 say "F7  -" get me_macros[1]
         @ 11,30 say "F8  -" get me_macros[2]
         @ 12,30 say "F9  -" get me_macros[3]
         @ 13,30 say "F10 -" get me_macros[4]
         read
         restscreen(9,29,14,53,me_save)
      case me_key == K_F7
         keyboard trim(me_macros[1])  // Insert first macro
      case me_key == K_F8
         keyboard trim(me_macros[2])  // Insert second macro
      case me_key == K_F9
         keyboard trim(me_macros[3])  // Insert third macro
      case me_key == K_F10
         keyboard trim(me_macros[4])  // Insert fourth macro
      endcase
   endif
   return ME_DEFAULT      // Process key's default value
```

Summary

After reading this chapter and working through the example, you should feel comfortable with the power available in Clipper's memo handling. You should know how to create memo variables, how to edit them, and how to write them back to disk. If you need to create a word processor in Clipper, you should be comfortable expanding the example application to meet your needs.

CHAPTER 24

Low-level File Access

A disk file may be viewed as a series of bytes stored together on the disk. The meaning and relationship of these bytes is determined by the application that uses the file. The file is interpreted according to the structure that a program imposes on it. Database files have their own structure, as do spreadsheet files, index files, word processing files, and so on.

Clipper provides a set of functions to allow access to any file in an unstructured, low-level manner. Data from the file can be read or written on a byte level. In this chapter we will cover these functions and provide some examples. We will also review the user-defined functions found in the FILEIO.PRG included in the CLIPPER5\SOURCE\SAMPLE directory.

DOS and Files

In earlier versions of DOS, information about disk files was passed to the operating system through a File Control Block. A File Control Block (FCB) is a 44-byte area of memory which contains descriptive information about the files. This information includes the file name and extension, the record size, the current record, the current block, and other data needed for the operating system to process the file.

In order to access files, your program had to change values in the FCB and create an area, called the Data Transfer Area, to hold information from the file. This buffer area was always the size of a single record from the file, allowing one record to be in memory at a time.

DOS versions 2.0 and above relegated the file information to the control of the operating system. The operating system creates a table, called the file handle table, which holds information needed to access files. All information in this table is hidden from us and maintained by DOS. When we open or create a file, DOS allocates an entry in the table to hold the new file's information. It returns to our program a numeric pointer indicating the entry in the table where the file information is located. This pointer is referred to as a file handle.

File handles

Any file opened by an application is assigned a file handle. A file handle is a sixteen-bit integer value used to point to an entry in a file table created by DOS. This value can range from zero through 65,535. When the FOPEN() or FCREATE() function is called successfully, a handle will be returned. This handle is then used for all subsequent access to the file. If either function call fails, a **-1** will be returned instead of a valid file handle.

Expanding the file handle table

DOS sets aside an area of memory for the file handle table. Each time a file is opened, its handle is added to this table. When the file is closed, the slot it occupied in the table becomes available again. The size of this table is determined by the operating system. Versions of DOS below 3.3 provide for a maximum of 20 file handles in the table. DOS versions 3.3 and above allow 255 entries in the table, although the default value is 20. You can increase the number of handles available. The method for doing so varies depending upon the version of DOS you are using.

DOS 3.3 and above. For DOS versions 3.3 and above, a statement may be included in the system configuration file (CONFIG.SYS) which indicates the size of the file handle table. Its syntax is:

```
FILES = <nHandles>
```

Adding this statement to the CONFIG.SYS file and rebooting will cause DOS to create a larger file handle table, making room for **<nHandles>** number of files. This new number becomes the maximum number of open files DOS will allow.

In addition, the Clipper application must be informed of the number of file handles. This is done through the use of the Clipper environment string. Its syntax is:

```
SET CLIPPER=F<nHandles>
```

Whichever value is lower will determine the number of handles available to the Clipper application. For example, the following environment will make 45 file handles available to the Clipper program.

```
CONFIG.SYS
FILES = 49

SET CLIPPER=F45
```

It is always better to use an odd number for the file handle setting in the CONFIG.SYS file. A bug in some DOS versions causes an extra 60K bytes of memory to be wasted if an even number is used.

DOS 3.2 and Below. Clipper does not provide direct support for more than twenty open files under DOS versions 3.2 and below. The distribution disk which accompanies this book contains a utility function that can be used to increase the number of open handles. This utility contains a UDF called Handles(). The syntax for the function is:

```
Handles( <nHandles_needed> )
```

A Clipper program is assigned a location in memory for the file handle table when it starts. The Handles() function increases the size of the file handle table by redirecting where DOS looks for it to a different, larger section of memory.

Handles used by DOS (the first five)

The first five entries in the handles table are used by the operating system. DOS automatically opens these five handles whenever a program starts. The five standard handles are listed in table 24.1.

Table 24.1 Standard DOS file handles

Handle	*Description*	*Default Device*
0	Std Input	(CON:) - Keyboard
1	Std Output	(CON:) - Screen
2	Std error output	(CON:) - Screen
3	Std auxiliary device	(AUX:) - Serial port (COM1)
4	Std printer	(PRN:) - Parallel port (LPT1)

If you are very tight on file handles, your program could close the fourth and fifth handles. The first three handles should never be closed.

Counting file handles

To determine the number of handles a program needs, we need to understand how file handles are assigned. Each database file opened during a program occupies at least one handle. If the file contains a memo field, a second handle is used for the DBT file. Each index file also uses one handle. It is easy to see that a large program can quite quickly exceed the available handles.

For example, assume a customer file has the following structure and two indexes:

CUSTOMER.DBF
```
Id_code     Char     10
Company     Char     25
City        Char     15
State       Char      2
Zip_code    Char     10
Notes       Memo     10
```

The first index is on **Id_code** and the second is on **Company**. The command

```
use CUSTOMER index cust_01,cust_02
```

will require four handles: one for the .DBF, one for the .DBT, and one for each index.

Opening files

To open any file in an unstructured or low-level mode, two functions are available. FCREATE() is used to create a new file and FOPEN() will open an existing file. Extreme care should be used with all low-level file functions. While the lack of an imposed structure provides tremendous flexibility, it also has potential dangers. If FCREATE() is called on an existing file, the file will be truncated. FCREATE() does not care that the file it just erased contained the last two years' worth of payroll data.

FCREATE()

FCREATE() opens a file for writing. If the file does not exist, it will be created. If the file already exists, it will be truncated to zero bytes with no error or warning given. The syntax is:

```
nhandle := FCREATE( <cFile_name>[,<nAttributes>] )
```

Always assign the output from FCREATE() to a variable, since this is the only way to refer to the file's handle.

The file specified will be created in the current DOS directory unless a path is specified in the file name. In addition, the file is assigned an attribute. The attribute indicates the type of file. The **<nAttributes>** may be one of the values in Table 24.2.

Table 24.2 File opening attributes

Value	*FILEIO.CH*	*Description*
0	FC_NORMAL	Read/write file, the default value
1	FC_READONLY	Read-only
2	FC_HIDDEN	Hidden, not seen by DOS commands
4	FC_SYSTEM	System, will also be hidden from DOS

The attributes may also be combined. For example, to create a file that is both hidden and read-only, we could set the attribute to:

```
FC_READONLY + FC_HIDDEN      =   1 + 2    =  3
```

A hidden system file would be:

```
FC_HIDDEN + FC_SYSTEM        =   2 + 4    =  6
```

The file is created in read-write mode, regardless of the attributes. The attributes are not applied to the file until the file is closed. This allows your program to create a read-only file and write data to it. When your program closes the file, it will be written to the directory as a read-only file if so specified.

The example function in listing 24.1 creates a CONFIG.SYS file if one does not exist. The parameter to the function indicates the number of file handles CONFIG.SYS should have.

Listing 24.1 NewCfgSys()

```
#include "FILEIO.CH"
#define CRLF chr(13)+chr(10)

function NewCfgSys(nHandles)
local fh
```

```
if !file("CONFIG.SYS")
   fh := fcreate( "config.sys",FC_NORMAL )
   if fh > -1
      fwrite(fh,"FILES="+str(nHandles,3)+CRLF )
      fclose(fh)
   else
      ? "DOS error occurred, ",ferror()
      return .f.
   endif
endif
return .t.
```

If the handle returned by FCREATE() is a **-1**, then a DOS error occurred when attempting to create the file. The FERROR() function will return the error number. The probable error codes that FCREATE() might return are: 3 (path not found), 4 (no more handles), or 5 (access denied). Access denied occurs if you are trying to create an existing read-only file or if there is no more room in the current directory for a file. See the error handling section of this chapter for more information.

FOPEN()

FOPEN() is used to open an existing file and provide low-level access to it. Its syntax is:

```
nHandle := FOPEN( <cFile_name> [,<nOpen_mode>] )
```

Be sure to assign the output from FOPEN() to a variable, since this is the only way to refer to the file's handle.

The file specified must be in the current DOS directory unless a path is specified in the file name. FOPEN() will not search the DOS path nor the Clipper SET DEFAULT setting.

The **<nOpen_mode>** may be one of the numbers in table 24.3. If the file cannot be opened in the requested mode, FOPEN() will return a **-1** error value. If you created a read-only file and later try to open it for read-write, the FOPEN() function will fail to open the file.

Table 24.3 File open modes

Value	*FILEIO.CH*	*Description*
0	FO_READ	Read-only mode, the default value
1	FO_WRITE	Write-only mode
2	FO_READWRITE	Open for reading/writing

For safety reasons, the default open mode is **0** — read only access. Other modes are possible although they are not documented in the user's guide. Any undocumented feature is subject to change, so use these modes at your own risk. These modes are present in earlier versions of Clipper as well. The additional modes are used when opening files on a network. The modes are listed in table 24.4.

Table 24.4 File sharing modes

Value	*Description*
0	Compatibility mode, the default
16	Deny read access/write access to others
32	Deny write mode to others
48	Deny read mode to others
64	Allow read/write access from others

The value should be added to the desired open mode from Table 24.3 to get the actual mode to use. For example, to open a file for write access and deny read mode, you would use an open mode of 49 (1 from Table 24.3 plus 48 from Table 24.4). The most common mode on a network would be 66, an open file for read/write access that allows others to read and write as well.

If you need to work with network files, you should use manifest constants instead of hard coding the modes. This will make changes easier if Nantucket changes the open modes in the future. Here are the manifest constants for the network open modes.

```
#define FS_DENY_ALL     16
#define FS_DENY_WRITE   32
#define FS_DENY_READ    48
#define FS_ALLOW_ALL    64
```

If the handle returned by FOPEN() is a **-1**, then a DOS error occurred when attempting to open the file. The FERROR() function will return the error number. The probable error codes that FOPEN() might return are: 2 file not found, 4 no more handles, 5 access denied or 12 invalid access. Invalid access occurs if you are trying to open a file on a network while someone else is using it and the file cannot be opened as shared. See the error handling section of this chapter for more information.

Moving about the file

The system maintains a file pointer for each file handle in the file table. The pointer is used to determine where the next read or write operation will occur.

FSEEK()

The FSEEK() function is the primary method used to move the file pointer. Its syntax is:

```
nPosition := FSEEK( <nHandle>,<nOffset>,<nOrigin> )
```

<nHandle> represents the entry in the file handle table you wish to move. **<nOffset>** indicates how many bytes the pointer should be moved. **<nOrigin>** indicates the starting position for the movement. It can be one of the three choices in Table 24.5.

Table 24.5 File origins

Value	*FILEIO.CH*	*Description*
0	FS_SET	Beginning of file
1	FS_RELATIVE	Current file pointer position
2	FS_END	End of file

The default value is zero or FS_SET, the beginning of the file.

For example, to open a file and append data to it, you could use this code fragment:

```
#include "FILEIO.CH"
fh := fopen("TEST.DAT",FO_READWRITE)
if fh > -1
   fseek(fh,0,FS_END)         // Position pointer at
endif                         // end of the file
```

FSEEK() returns the file position after the pointer is moved. The return value from the function call

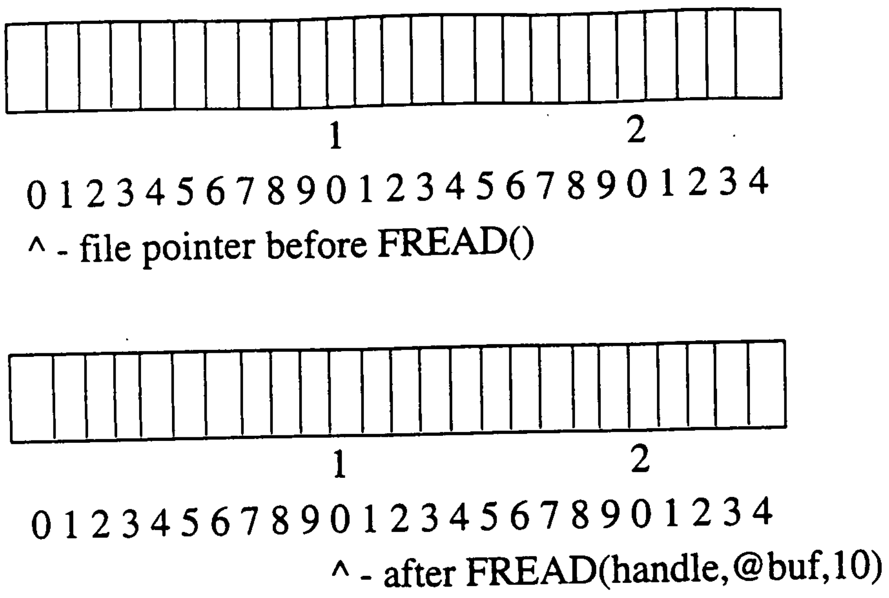
```
fseek(fh,0,2)
```

would be the last byte in the file, or the file's size. Be sure to keep the pointer in mind when writing data to the file. The lack of automatic error checking by the low-level file functions provides great flexibility but requires careful use of the functions.

The file reading and writing functions will also move the file pointer. The pointer will be moved from its current position to the current position plus the number of bytes read or written.

The example in Figure 24.1 shows the file pointer starting at zero, the beginning of the file, and being moved ten bytes after the call to FREAD().

Figure 24.1 File pointer movement



Reading information from files

There are two functions available to extract data from a file using a handle. These functions are FREAD() and FREADSTR(). The difference between the two is that FREADSTR() will only read up to a character zero (CHR(0)) while FREAD() will read any character, including CHR(0).

FREAD()

For more low-level files, FREAD() is preferable because many files use CHR(0) as a data byte. While it requires a bit more setup to use, it will return all the data from the file. The syntax for FREAD() is:

```
cbuffer     := space(<nBytes>)
nBytes_read := FREAD(<nHandle>,<@cBuffer>,<nBytes>)
```

The character variable to hold the bytes must be filled with spaces before FREAD() is called. It must be passed to the function by reference, hence the @ is required.

The return value is the number of bytes that were read. If this number does not agree with the number of bytes that were requested then either you've attempted to read past the end of the file or a DOS error occurred. You can check FERROR() to determine what error, if any, occurred.

The function in Listing 24.2 illustrates a file copy which creates a new file from the original file converted to uppercase.

Listing 24.2 Uppercase file copy

```
#include "FILEIO.CH"
#define   BUFSIZE 2048

function Ucopy(old_file, new_file)
local in_handle, out_handle, buffer := space(BUFSIZE),nbytes
in_handle := fopen(old_file,FO_READ)
if in_handle > -1
   out_handle := fcreate(new_file,FC_NORMAL)
   if out_handle > -1
      do while (nBytes := fread(in_handle, @buffer, BUFSIZE)) > 0
         fwrite(out_handle,upper(buffer),nBytes)
         buffer := space(BUFSIZE)
      enddo
      fwrite(out_handle, upper(buffer), nBytes)
      fclose(in_handle)
      fclose(out_handle)
   else
      ? "Error ",ferror()," creating file ",new_file
   endif
else
   ? "Error ",ferror()," opening file ",old_file
endif
return nil
```

FREADSTR()

The FREADSTR() function reads bytes from a file into a memory variable. Since a Clipper string is terminated by a NULL character, CHR(0), the function will read either the requested number of bytes or up to an occurrence of a CHR(0). Its syntax is:

```
<cBuffer> := FREADSTR(<nHandle>,<nBytes>)
```

Regardless of which reading function you use, reading should be done into larger memory variables rather than into just a few bytes. Most hard disks have sector sizes of 2048 bytes (2K) which means that 2048 bytes will be read into memory by the operating system regardless of the number of bytes you specify. If you perform your reads in smaller byte sizes, the program will be forced to do extra disk reads. A disk read slows the performance down dramatically and so should be limited to the minimal number needed to get the job done. See the section in this chapter on speeding up low-level access for more information on buffering your read operations.

Writing bytes into files

The FWRITE() function is used to write information into a file identified by its handle. All information to be written to the file must be converted to character form before writing. If a line of text is to be written, a delimiter such as a carriage return/ line feed pair should be written to the file as well.

FWRITE()

The FWRITE() function is used to write a character string into a file at the current file position. Its syntax is:

```
nBytes_out := FWRITE(<nHandle>,<cBuffer>,<nWritebytes>)
```

The **<nHandle>** refers to a file which must be opened for writing or in read/write mode. The **<cBuffer>** contains the actual characters to be written, and the **<nWritebytes>** represents the number of characters to write. If the number is not specified, it will default to the length of the **<cBuffer>** string.

The function will return the number of bytes written into the file. If this return value does not agree with the number of bytes then some sort of error occurred during the write process. FERROR() will return a code indicating why the write failed. The most likely reason for a failure during the FWRITE() would be lack of sufficient disk space.

The function in listing 24.3 illustrates using FWRITE() to create a log of information about users. It might be called after the user logs on to the program or performs a particular module or function.

Listing 24.3 Writelog()

```
#define USERLOG "USER.LOG"
#include "FILEIO.CH"

function Writelog(cUserid,cMessage)
local fh,buf,retval:=.t.
if file("USER.LOG")
   fh:= fopen(USERLOG,FO_WRITE)
   if fh > -1
      fseek(fh,0,FS_END)      // Move to end of file
   else
      retval := .f.
   endif
else
   fh := fcreate(USERLOG,FC_NORMAL)
endif
```

```
if fh > -1
   buf := cUserid +" "+ ;
          dtoc( date() )+" "+;
          time()+" "+cMessage+chr(13)+chr(10)
   if fwrite(fh,buf,len(buf)) <> len(buf)
      retval :=.f.
   endif
   fclose(fh)
else
   retval:=.f.
endif

return retval
```

Closing the file

Once a file handle is no longer needed by the program, the program should close that handle. If the file was created with FCREATE(), the file attribute designated by FCREATE() (see Table 24.2) will be applied to the file by the FCLOSE() function. The syntax to close the file is:

```
<logical> := FCLOSE( <nHandle> )
```

<nHandle> represents the handle number returned from a previous FCREATE() or FOPEN() function call. FCLOSE() will first flush the DOS buffers to disk and then attempt to close the file. It returns a logical true if the buffers were flushed and the file was closed. If a problem occurred, a false will be returned.

Error handling

If an error occurs using one of the low-level functions, the function will return an error indicator. In addition, FERROR() will contain a non-zero number. Table 24.6 lists the return values that the low-level functions use to indicate an error was detected.

Table 24.6 Low-level error return values

Function	*Error return value*
FCLOSE()	.F.
FOPEN()	-1
FCREATE()	-1
FREAD()	Bytes read <> Bytes requested
FREADSTR()	Bytes read <> Bytes requested
FWRITE()	Bytes written <> Bytes requested

FERROR()

Once an error has been detected, the FERROR() function will return a numeric code indicating what error has occurred. Its syntax is simply:

```
<nErrorno>  := FERROR()
```

If no error was encountered, FERROR() will return an error code of zero. Table 24.7 lists possible FERROR() values and the course of action to recover from them.

Table 24.7 DOS errors

Error	*Meaning*	*Possible solutions*
0	Successful	
2	File not found	Check spelling of file name.
3	Path not found	Check spelling of path name, also test to see if the path exists.
4	Too many files	Increase your file handles size.
5	Access denied	Make sure the file is not set to read-only. On a network, check that you have access rights to the file.

Error	*Meaning*	*Possible Solutions*
6	Invalid handle	Check the value of the variable to be sure a handle was opened or created properly.
15	Invalid drive	The drive designated does not exist.
19	Write protected	The diskette is write protected.
21	Drive not ready	Usually because the drive door is open.
29	Write fault	Probably the disk is full or, on a network, the user doesn't have write privileges to the directory.
32	Share violation	The file is currently being used by another user and your attempt to access the file is not permitted.

The most common FERROR() error return values are two through five, since your program may not be able to check the DOS environment of all the files to which it needs access.

FERROR() can be useful to determine the status of various directories and drives. For example, the code below uses DOS's NUL: device to check a drive status. The NUL: device is a standard DOS device that can be used much like any other device, but will not be saved anywhere. It is most frequently used to redirect output from DOS operations to never-never land. The function attempts to open a NUL: device on a particular drive. This will always fail when using the low-level functions. If FERROR() returns a three, the function failed because the drive is not valid. The Isdrive() function will return true if the drive letter specified appears to be valid or false if the drive is not available or ready. The function can easily be adapted to handle subdirectories as well.

```
function Isdrive(drv_letter)
local fh := Fopen(drv_letter+":\NUL:")
return( ferror() <> 3 )
```

FERROR() can also be used to check if a file can be written to. The Ok2write() function shown below returns true if the file can be opened in a read-write mode and false if not. By checking for write access, your program can detect that a memory variable file cannot be saved. Hopefully, you will check it before your user has changed all the values and wants to save them to disk.

```
#include "FILEIO.CH"

function Ok2write(cFile)
local retval:=.f.,fh
if file( cFile )
   fh     := fopen( cFile,FO_READWRITE )
   retval := (fh >= 0 .and. ferror()== 0 )
   fclose(fh)
endif
return retval
```

Error recovery

If your program detects an error, it should provide some guidance to assist the user in recovering from the error. The program in listing 24.4 illustrates a program to be called if the value of FERROR() is not zero. It will present a message to the end-user advising him or her how to correct the problem.

Listing 24.4 Errmsg()

```
Function Errmsg(nErr)
do case
case nErr == 2
   ? "Check the spelling of the file name...."
case nErr == 3
   ? "Check that the proper path is designated..."
case nErr == 4
   ? "Need to increase CONFIG.SYS files statement..."
case nErr == 5
   ? "The file is probably set to read-only...."
case nErr == 6
   ? "Programming problem - contact the programmmer..."
```

```
case nErr == 15
   ? "You've specified an invalid drive letter...."
case nErr == 19
   ? "Remove the write-protect tab from the diskette..."
case nErr == 21
   ? "Try shutting the drive door..."
case nErr == 29
   ? "The disk is full....."
case nErr == 32
   ? "Someone else is using the file..."
case nErr <> 0
   ? "CONTACT the programmer immediately!!!!"
endcase
return nil
```

Extending the file functions, FILEIO.PRG

The Clipper disks include a set of user-defined functions to enhance the low-level file tools. These functions are found in FILEIO.PRG which can be found in the \CLIPPER5\SOURCE\SAMPLE directory. The program should be compiled with the following switches:

/n	- No default procedure name
/w	- Enable warnings
/m	- Compile module only
/a	- Automatic **memvar** declaration

The sample functions in FILEIO.PRG can be grouped into two sections: text-file functions and file status functions.

Text file functions

A text file is a file that is broken down into lines. A line is a series of characters terminated by some delimiter, usually a carriage return or a carriage return/line feed pair. Your Clipper .PRG files, CONFIG.SYS, and AUTOEXEC.BAT files are a few examples of text files.

Freadln()

Freadln() reads a single line or multiple lines from a text file into a character buffer. Its syntax is:

```
cString := Fradln(<nHandle>,<nLines>,<nLength>,<cDelimiter>)
```

A line is usually delimited by the carriage return/line feed pair. This will be the default delimiter if none is supplied. The default number of lines is 1 and the default line length is 80 characters.

The Fgets() function included in FILEIO.PRG is identical to the Freadln() function. The Fgets() syntax is based upon a similar function available in FoxPro. You could create a preprocessor #xtranslate definition if you wanted to eliminate the Fgets() overhead from the FILEIO program. See Chapter 7 for more information about the preprocessor.

Fwriteln()

The Fwriteln() function is used to write a line of text to a file. The text is written, followed by a carriage return/line feed pair. Its syntax is:

```
nBytes :=FWRITELN(<nHandle>, <cString>, <nLength> )
```

There is no default line length in this function, so be sure to pass a value. The string will have a carriage return/line feed pair written to the file after it, so the **<nLength>** should include two extra bytes to accommodate the delimiter.

The Fputs() function included in FILEIO.PRG is identical to the Fwriteln() function. (It is also a function from FoxPro.) You could create a preprocessor #xtranslate definition if you wanted to eliminate the Fputs() overhead from the FILEIO program. See Chapter 7 for more information about the preprocessor.

Example program

The text file functions can be used to update any text file. In this example, a function called Fix_cfg() will update the values in the CONFIG.SYS file to the proper number of files and buffers. Its syntax is:

```
<logical>  := Fix_cfg( <nFiles>,<nBuffers> )
```

The **<nFiles>** indicates the number of file handles the CONFIG.SYS file should have. **<nBuffers>** indicates the number of buffers. If the CONFIG.SYS file has at least these numbers, a true value will be returned. If not, the file will be corrected and a value of false will be returned. Presumably your application program will then ask the user to reboot the computer and run the program again.

The Fix_cfg() function is shown in listing 24.5.

Listing 24.5 Fix_cfg()

```
*
* Be sure program has access to FILEIO.PRG functions
*
#include "FILEIO.CH"
#define  CRLF chr(13)+chr(10)

function Fix_cfg(nFiles, nBuffers)
local fh, retval := .t., cfg_array := {},buf := "", x, k, x2
local numfiles := 0, numBufs := 0
if file("CONFIG.SYS")
   fh := fopen("CONFIG.SYS",FO_READWRITE)
   if fh > -1
      buf := upper(Freadln(fh,1,80))
     do  while !empty(buf)
         x := at("FILES",buf)
```

```
            if x > 0
               x2        := at("=",buf)
               numFiles := val(substr(buf,x2+1,25))
            endif
            x := at("BUFFERS",buf)
            if x > 0
               x2        := at("=",buf)
               numBufs  := val(substr(buf,x2+1,25))
            endif
            aadd(cfg_array,buf)
            buf := Freadln(fh,1,80)
         enddo
         if numFiles < nFiles .or. numBufs < nBuffers
            fclose(fh)
            frename('config.sys', 'config.old')
            fh :=fcreate("CONFIG.SYS",FC_NORMAL)
            for k := 1 to len(cfg_array)
                x := at("FILES=",cfg_array[k])
                if x > 0
                   cfg_array[k] := trim("FILES="+str(nFiles,3))
                endif
                x := at("BUFFERS=",cfg_array[k])
                if x > 0
                   cfg_array[k] := trim("BUFFERS="+str(nBuffers,3))
                endif
               Fwriteln(fh,cfg_array[k],len(cfg_array[k])+2)
            next
            retval :=.f.
         endif
         fclose(fh)
      endif
   else
      fh := fcreate( "CONFIG.SYS",FC_NORMAL )
      if fh > -1
         fwrite(fh,"FILES="+str(nFiles,3)+CRLF+;
                   "BUFFERS="+str(nBuffers,3)+CRLF)
         fclose(fh)
      endif
      retval :=.f.
   endif
   return retval
```

File status functions

The file status functions return information about the file size and location as well as positioning the file.

Filetop()

The Filetop() function moves the file pointer to the first byte in the file. Its syntax is:

```
< nPosition> := FileTop( <nHandle> )
```

The **<nPosition>** that is returned indicates the new file pointer position, which will always be zero. Filetop() is equivalent to writing

```
Fseek(<nHandle>,0,0)
```

FileBottom()

The Filebottom() function moves the file pointer to the last byte in the file. Its syntax is:

```
<nPosition> := Filebottom( <nHandle> )
```

The **<nPosition>** that is returned indicates the new file pointer position, which will always be the size of the file. Filebottom() is equivalent to writing

```
Fseek(<nHandle>,0,2)
```

FilePos()

The FilePos() function returns the current value of the file pointer. Its syntax is:

```
<nPosition> := FilePos( <nHandle> )
```

The **<nPosition>** that is returned indicates the current file pointer position. FilePos() is equivalent to writing

```
Fseek(<nHandle>,0,1)
```

Preprocessor #xtranslate

The functions Filetop(), Filebottom(), and Filepos() can also be written as preprocessor commands using the #xtranslate syntax. The listing in 24.10 shows a .CH header file that could be used to replace these three functions.

Listing 24.10 Preprocessor

```
#xtranslate FILETOP( <n> )    => Fseek( <n>,0,0 )
#xtranslate FILEBOTTOM( <n> ) => Fseek( <n>,0,2 )
#xtranslate FILEPOS( <n> )    => Fseek( <n>,0,1 )
```

FileSize()

The Filesize() function returns the number of bytes in the file and keeps the file pointer where it is positioned. Its syntax is:

```
<nBytes> := FileSize( <nHandle> )
```

The file size can be determined using the FSEEK() function. This function saves the current file pointer, determines the file size, and then returns to the same spot in the file. An example of its use could be:

```
if filesize(fh) > Diskspace(1)
  ? "Cannot fit file onto drive A:"
endif
```

Feof()

Feof() returns a logical value of true if the file pointer is positioned at the end of the file or false if it is not. Its syntax is:

```
<logical> := Feof( <nHandle> )
```

The function works by comparing the file pointer with the value returned from the Filesize() function. It is useful when reading a file to determine if the end of the file has been reached or not. For example:

```
do while !feof(fh)
   ? freadln(fh,1,80)
enddo
```

Conversion functions

When using the database drivers provided in Clipper's libraries, all data extracted from the .DBF file is converted by Clipper into its internal memory variable format. The low-level file functions do not have this conversion done for them. Clipper considers all data read from a low-level file to be character data. In many files, this is not the case. For example, Lotus spreadsheets store real numbers as illustrated in Figure 24.2.

Figure 24.2 - Lotus real number format

Format Info	Column Number	Row Number	Numeric value stored in IEEE long real format
1-Byte	2-Bytes	2-Bytes	8-Bytes

For Clipper to provide meaningful access to this data, it has to convert it from the character format to its numeric meaning. The conversion function to use depends upon the number of bytes the number occupies.

One byte

If one byte is extracted from a file, Clipper will return the character that the byte corresponds to on the ASCII chart. If that one byte should be treated as a number, the ASC() function can be used to perform the conversion. If a single byte number in the range of zero through 255 needs to be written to a file, you may use the CHR() function to convert the number to a character string for FWRITE() to use.

ASC()

The ASC() function takes a single character and returns its ranking on the ASCII chart. Its syntax is:

```
<nByte> := ASC( <cChar> )
```

For example, ASC("A") would return the number 65. See the ASCII chart in Appendix A for more examples.

CHR()
The CHR() function takes a single byte integer and returns its character equivalent from the ASCII chart. Its syntax is:

```
<cChar> := CHR( <nByte> )
```

For example, CHR(65) would return the character A. See the ASCII chart in Appendix A for more examples. The application program will probably view this byte as the number 65, but Clipper must convert it to a character format to write it to the low-level file.

Two bytes
If a two byte integer is to be extracted from a file, you may use the BIN2I() or the BIN2W() function. The BIN2I() function handles signed integers and the BIN2W()handles unsigned integers. In C, this two byte integer is referred to as **int** or **short**.

A two-byte integer consists of sixteen bits. For a signed integer, the first bit indicates a positive (bit=0) or negative (bit=1) value. This, in effect, reduces the number of bits from sixteen to fifteen. In an unsigned integer, all sixteen bits are used to represent the number. The BIN2I() function instructs Clipper to interpret the first bit as a sign-bit. The BIN2W() function causes the bit to be considered part of the numeric value.

BIN2I()
The BIN2I() function takes a two-character string and returns the integer equivalent of the string. This function is used for signed integers in the range of -32,768 to +32,767. Its syntax is:

```
<nInteger> := BIN2I( <cSigned_int> )
```

BIN2W()

The BIN2W() function takes a two-character string and returns the integer equivalent of the string. Unlike the BIN2I() function, BIN2W() will not return a negative number. Its range of numbers is zero through 65,535. The syntax for the function is:

```
<nInteger> := BIN2W( <cUnsigned> )
```

I2BIN()

The integer to binary function works in reverse of the BIN2x() functions. It will take an integer and convert it to a binary representation. Its syntax is:

```
<cShort> := I2BIN( <nInteger> )
```

In addition to being used for reading information from non-DBF files, these functions can be used to compress integer storage space. If the integer is in the ranges above, 2-3 bytes of storage can be saved per integer through use of these functions.

Four bytes

Some programs store large integers in a four-byte format. Clipper provides conversion features to handle this double integer format. In C, this format is called the **long integer**. Although the long integer may be signed or unsigned, the Clipper functions work with **signed long integers** only.

BIN2L()

The BIN2L() function takes a long integer represented as a four-character string and returns the integer equivalent of the string. Its range of values is -2,147,483,648 to +2,147,483,647. The syntax for the function is:

```
<nInteger> := BIN2L( <cLong_int> )
```

Bin2ul()

The Bin2ul() user-defined function takes a long integer represented as a four-character string and returns the integer equivalent of the string. Its range of values is zero to +2,147,483,647. The syntax for the function is:

```
<nInteger> := Bin2ul( <cLong_int> )
```

Listing 24.11 shows the the Bin2ul() user-defined function.

Listing 24.11 Bin2ul()

```
function Bin2ul(cLong)
local jj:=Bin2l(cLong)
if jj < 0
   jj := abs(jj)+ 2147483647
endif
return jj
```

L2BIN()

The long integer to binary function works in reverse of the BIN2L() function. It will take an integer and convert it to a long integer binary representation. Its syntax is:

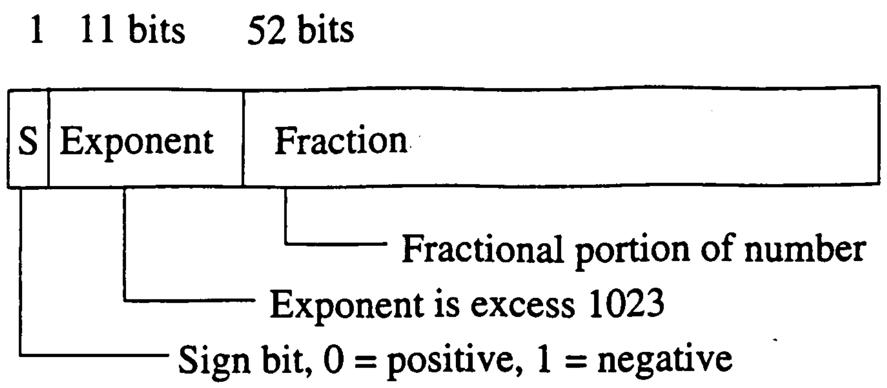
```
<cLong_int> := L2BIN( <nInteger> )
```

The long integer format can also be used to compress data storage for very large integers. The equivalent of ten numeric spaces can be represented in a four-byte string, for a 60% savings in storage space.

IEEE long real format

Some programs, such as Lotus and Btrieve, store real numbers in the IEEE long real format. This is a 64-bit (8 bytes) format as shown in Figure 24.3. This format is referred to as a **double** in C.

Figure 24.3 IEEE long real number format


1 11 bits 52 bits
S
Exponent
Fraction
Fractional portion of number
Exponent is excess 1023
Sign bit, 0 = positive, 1 = negative

sign	= if first bit is set, -1 else 1
exponent	= bits 2-11 after string is built
fraction	= bits 13-52 after string, plus an implied 1 bit at the front

For Clipper to work with an IEEE real number, you must convert it from an 8-byte string into a Clipper numeric. Clipper does not provide any built-in functions to perform this conversion.

Real2iee

Listing 24.6 contains a function that will take a real number and convert it to the IEEE long real format expressed as a string variable. Its syntax is:

```
<cIeeereal> := Real2iee( <nReal_number> )
```

Listing 24.6 Real2iee()

```
function Real2iee(nReal)
local retval:=replicate(chr(0),8), exponent, sign, j
if nReal<>0                             // If non-zero
   sign     := if(nReal<0,128,0)        // Set the sign bit
   exponent:= 1075               // Offset of 1023 + 52 bits max
   nReal    := abs(nReal)        // We already know the sign
```

```
      do while nReal < 2**52        // 2 raised to 52 power
         nReal *= 2                 // Multiply number by 2
         exponent--                 // Subtract one from exponent
      enddo
      do while nReal > 2*4503599627370496-1
         nReal /= 2
         exponent++
      enddo
      nReal  := int(nReal)   // Number is now an integer
      retval := ''           // Build the first 48 bits in reverse
      for j  := 1 to 6       // by dividing by 256 and writing
        retval += chr(nReal%256)
        nReal  := int(nReal/256) // remainder to string nReal
      next
      nReal  -= 16                  // Strip out four high end bits
      retval += chr(nReal +16 * (exponent%16))+;
                chr(sign+int(exponent/16))
   endif
   return retval
```

Iee2real

Listing 24.7 contains a function that will take a character string and convert it to a real number. Its syntax is:

```
<nReal_number> := Iee2real( <cIeeereal> )
```

Listing 24.7 Iee2real()

```
function Iee2real(cIee)
local retval:=0, k, x1 := "", sign, exponent, fraction
if cIee <> repl(chr(0),8)       // Are all bits set to zero?
   for k:=len(cIee) to 1 step -1
      x1 += Dec2bin(asc(substr(cIee,k,1)))
   next
   sign     := (substr(x1,1,1)=="1")
   exponent := Bin2dec(substr(x1,2,11)) - 1022
   fraction := Bin2dec("1"+substr(x1,13),.t.)
   retval   := if(sign,-1,1)* (fraction * (2**exponent))
endif
return retval
```

```
function Bin2dec(_string)
local retval :=0 ,k
for k:=1 to len(_string)
  retval := if(substr(_string,k,1)=="0",0,1)+retval+retval
next
return retval

function Dec2bin(_number)
local tmp := _number, retval := "", remd, quot
do while .t.
   quot  := int(tmp/2)
   remd  := abs(tmp)-2*abs(quot)
   retval:= substr("01",remd+1,1)+retval
   if quot== 0
      exit
   endif
   tmp := quot
enddo
do while len(retval) <16
   retval := "0"+retval
enddo
return retval
```

Both of these functions can be sped up dramatically through the use of C functions. The C equivalent for these functions is illustrated in listing 24.8.

Listing 24.8 Ieereal.c

```
 * syntax:   ieereal( nIEEVar )
 * returns:  char string of length 8. each char in string
 *           represents the appropriate byte of nIEEVar.
 *
 * syntax:   realiee( cIEEStr )
 * returns:  double. converts cIEEStr to its double equivalent
 *
 * compile:  msc 5.1 - cl /W3 /Ox /AL /Gs /c /FPa /Zl ieereal.c
 *
 */

#include "extend.h"

CLIPPER ieereal(void);
CLIPPER realiee(void);


union {
         double n;
         char   s[8];
       } v;


CLIPPER ieereal()
{
    double n;

    n = _parnd(1);

    _retclen( (char far *) &n, 8);
}


CLIPPER realiee()
{
    char i;
    char far *t;

    t = _parc(1);

    for (i = 0; i < 8; i++)
        v.s[i] = *t++;

    _retnd( v.n );
}
```

Special thanks to Mike Taylor who provided these functions.

Speeding up low-level access

The low-level file functions work directly with disk files. You can improve speed dramatically by creating a buffer and doing read and write operations only when the buffer is full. Since disk operations are among the slowest the computer performs, the fewer we use, the better.

Listing 24.9 illustrates the low-level functions extracting information from a .DBF file header. (See the .DBF structure in Chapter 18). This function extracts the information one byte at a time. The result is seven disk reads.

Listing 24.9 Readdbf1

```
* Program...: Readdbf1
*
parameter dbf_file
local fh, one_byte := " ", vers_id, yy, mm, dd
local num_recs, hdr_size, rec_size

fh := fopen(dbf_file,0)      // Open the file for read only
if fh > -1                   // File opened ok?
   fread(fh,@one_byte,1)
   vers_id := asc(one_byte)
   fread(fh,@one_byte,1)
   yy      := asc(one_byte)
   fread(fh,@one_byte,1)
   mm      := asc(one_byte)
   fread(fh,@one_byte,1)
   dd      := asc(one_byte)
   one_byte:= space(4)
   fread(fh,@one_byte,4)
   num_recs:= bin2l(one_byte)
   one_byte:= space(2)
   fread(fh,@one_byte,2)
   hdr_size:= bin2w(one_byte)
   fread(fh,@one_byte,2)
   rec_size:= bin2w(one_byte)
   fclose(fh)
endif
```

Note also the use of the ASC(), BIN2L(), and BIN2W() functions to convert the character data read into numeric information.

Listing 24.10 performs the same function as Listing 24.9 but uses only one disk read. While the savings on this small example will not appear to be dramatic, they will add up on any large amounts of file I/O.

Listing 24.10 Readdbf2

```
* Program...: Readdbf2
*
parameter dbf_file
local fh, buffer := space(12), vers_id, yy, mm, dd
local num_recs, hdr_size, rec_size

fh := fopen(dbf_file,0)      // Open the file for read only
if fh > -1                   // File opened ok?
   buffer   := fread(fh,@buffer,12)
   vers_id := asc(substr(buffer,1,1))
   yy      := asc(substr(buffer,2,1))
   mm      := asc(substr(buffer,3,1))
   dd      := asc(substr(buffer,4,1))
   num_recs:= bin2l(substr(buffer,5,4))
   hdr_size:= bin2w(substr(buffer,9,2))
   rec_size:= bin2w(substr(buffer,11,2))
   fclose(fh)
endif
```

By using a buffer we can reduce the number of disk reads and/or disk writes. This is the same technique to improve speed that many disk cache programs utilitize. If your program is going to spend a lot of time reading from a file, you should read the file in large buffers and parse it out as needed.

If you plan on reading large amounts of data from a file, the functions in listing 24.11 can be used to buffer your read and write operations, substantially speeding up the application. Be sure to specify the /N switch when you compile the functions. The two functions are Bread() and Bwrite().

Bread() — buffered read

This function takes a numeric parameter and returns that number of bytes from the buffer. If there are not enough bytes in the buffer to fulfill the read request, a new buffer will be read in from the disk. Its syntax is:

```
<cString>  := Bread(<nHandle>,<nBytes>)
```

Bwrite() — buffered write

This function takes a character string and adds it to the buffer. If the buffer size exceeds some constant number, the buffer is written to the disk and cleared out for the next set of data. Its syntax is:

```
<logical> := Bwrite(<nHandle>,<cString>)
```

Bwrite() will return false if the disk write failed. In this case, check FERROR() to determine why the write failed. All other times, Bwrite() will return a true value.

Listing 24.11 Buffered Read/Write

```
#define BUFFER_SIZE  4096
static buffer := ""

function Bread(nHandle,nBytes)
local tmp
```

```
if nBytes > len(buffer)
   tmp := buffer
   buffer := space(BUFFER_SIZE)
   fread(nHandle,@buffer,BUFFER_SIZE)
   buffer += tmp
endif
tmp := substr(buffer,1,nBytes)
buffer := substr(buffer,nBytes+1,BUFFER_SIZE)
return tmp

function Bwrite(nHandle,cString)
local x := .t.
buffer += cString
if len(buffer) >= BUFFER_SIZE
   x := (fwrite(nHandle,buffer)==len(buffer))
   buffer :=""
endif
return x
```

Sample programs

The sample programs illustrate how the low-level functions can be used to work with a variety of files.

ZipDir()

The Zipdir() function in listing 24.12 can be used to read the file directory table from a zipped file. Zip is a data compression program created by Phil Katz and is a common tool on many systems. Zip can compress database files as much as 90% which reduces disk storage space and file transfer time. A single zip file can also hold many individual files in their compressed form. It does this by maintaining a directory of files. The Zipdir() function in listing 24.12 uses the low-level functions to search for this directory and returns an array containing the file names it finds there.

Listing 24.12 ZipDir()

```
#include "FILEIO.CH"

function ZipDir(zip_file)
local zip := fopen(zip_file,FO_READ), tmp :=space(4)
local method, orig_date, fn_date, comp_size, uncomp
local fn_size, extra, junk, _array:={}, file_name
do while .t.
   tmp :== space(4)
   if fread(zip,@tmp,4) <> 4    // End of file
      exit
   endif
   if tmp = "PK"+chr(1)+chr(2)  // Central dir
      exit
   elseif tmp <> "PK"+chr(3)+chr(4)
      exit
   endif
   tmp := space(26)
   fread(zip,@tmp,26)
   method    := bin2i(substr(tmp,5,2))
   orig_date := bin2i(substr(tmp,9,2))
   fn_date   := Dos_date(orig_date)
   comp_size := bin2l(substr(tmp,15,4))
   uncomp    := bin2l(substr(tmp,19,4))
   fn_size   := bin2i(substr(tmp,23,2))
   extra     := bin2i(substr(tmp,25,2))
   file_name := freadstr(zip,fn_size)
   tmp       := space(extra)
   junk      := if(extra==0,"",fread(zip,@tmp,extra))
   fseek(zip,comp_size,1)
  aadd(_array,{file_name,comp_size,uncomp,fn_date})
enddo
fclose(zip)
return _array
```

```
function Dos_date(_datestamp)
local temp :=Dec2bin(_datestamp), yy, mm, dd
yy := Bin2dec(substr(temp,1,7)) +1980
mm := Bin2dec(substr(temp,8,4))
dd := Bin2dec(substr(temp,12,5))
return ctod(str(mm,2)+"/"+str(dd,2)+"/"+str(yy,4))

function Bin2dec(_string)
local retval :=0 ,k
for k :=1 to len(_string)
   retval := if(substr(_string,k,1) == "0",0,1)+retval+retval-
next
return retval

function Dec2bin(_number)
local tmp := _number,retval:="",remd,quot
do while .t.
   quot  := int(tmp/2)
   remd  := abs(tmp)-2*abs(quot)
   retval:= substr("01",remd+1,1)+retval
   if quot == 0
      exit
   endif
   tmp := quot
enddo
do while len(retval)<16
   retval := "0"+retval
enddo
return retval
```

Zipdir() will return an array of arrays. Each file within the zip directory will contain a four element array with the structure shown in Table 24.8.

Table 24.8 Zipdir() directory structure

Element	*Contents*
1	DOS file name
2	Size of compressed file

Element	*Contents*
3	Original DOS file size
4	DOS file date stamp

DBF2DIF()

DIF (Data Interchange Format) is a program-independent method of storing disk information. A .DIF file contains a header indicating the file name, number of fields, and number of records. It is followed by a data section which contains the actual data. Table 24.9 illustrates the format for a .DIF file.

Table 24.9 DIF file format

Record	*Contents*
1	TABLE <cr/lf> 0,1 <cr/lf> DIF name <cr/lf>
2	VECTORS <cr/lf> 0,<columns> <cr/lf> "" <cr/lf>
3	TUPLES <cr/lf> 0,<rows> <cr/lf> "" <cr/lf>
4	DATA <cr/lf> 0,0 <cr/lf> "" <cr/lf>
	Data section starts here
Each new row	-1,0 <cr/lf> "BOT" <cr/lf>
End of file	-1,0 <cr/lf> "EOD" <cr/lf>

The program in Listing 24.13 reads a .DBF file and converts it into a .DIF file by using the low level file functions. The parameters to the function are the name of the .DBF file to be read and the name of the .DIF file to be created. Its syntax is:

```
DBF2DIF( <cDbf_file>,<cDIF_file> )
```

The function will return a zero if the file was successfully created or will return the DOS error code if an error occurred.

Listing 24.13 DBF2DIF()

```
#define SQ chr(34)
#define DQ chr(34)+chr(34)
#define CRLF chr(13)+chr(10)

function DBF2DIF(cDbf,cDif)
local retval := 0, buffer, ctr := 0,k,mfld
local how_many, fh, nbytes

use (cDBF) new ALIAS input
go top
fh := Fcreate(Cdif,0)
if fh > -1
  buffer :='TABLE'+CRLF+'0,1'+CRLF+SQ+cDBF+SQ+CRLF +;
           'VECTORS'+CRLF+'0,'+str(fcount(),2)+CRLF+DQ+CRLF+;
           'TUPLES'+CRLF+'0,'+str(lastrec(),6)+CRLF+DQ+CRLF+;
           'DATA'+CRLF+'0,0'+CRLF+DQ+CRLF
   nbytes := fwrite(fh,buffer,len(buffer))      // DIF header
   if nbytes <> len(buffer)
      retval :=ferror()
      fclose(fh)
      return retval
   endif
   how_many := fcount()
   do while !eof()
      ctr++
      buffer := '-1,0'+CRLF+'BOT'+CRLF
      for k := 1 to how_many
        mfld   :=Fieldget(k)
        buffer += DIFPUT(mfld)
      next
      Fwrite(fh,buffer)
      skip +1
   enddo
   fwrite(fh,'-1,0'+CRLF+'EOD'+CRLF,11)
   retval := ferror()
   fclose(fh)
else
   retval := ferror()
endif
return retval

function DIFPUT(fld_data)
local fld_type := valtype(fld_data), retval := ""
if fld_type == "C"
   retval := '1,0'+CRLF+SQ+trim(fld_data)+SQ+CRLF
elseif fld_type == "L"
   retval := '0,'+if(fld_data,'1','0')+CRLF+;
             SQ+if(fld_data,'true','false')+SQ+CRLF
elseif fld_type == "N"
   retval := '0,'+ltrim(trim(str(fld_data)))+CRLF+SQ+'V'+SQ+CRLF
elseif type("fld_data") == "D"
   retval := '1,0'+CRLF+SQ+dtoc(fld_data)+SQ+CRLF
endif
return retval
```

The .DIF file was the file format used by Dan Bricklan's VisiCalc program (the first spreadsheet program on a personal computer). Spreadsheet programs have since defined their own formats, but many programs still can create and read .DIF files.

Summary

After reading this chapter you should feel comfortable working with the low-level file functions and you should be familiar with the terminology describing low-level files. You should have a healthy respect for the potential of these functions, in terms of both use and abuse. Based upon our example, you should be eager to try to create various files from within a Clipper program. If you can determine the structure of a file, Clipper's functions can make that file available to you.

Listing 13.3 Validating a Character-Type Date

```
theDate := "  /  /  "
@ 10, 12 say "The date " get theDate picture "99/99/99"
read
theDate := dtoc(ctod(theDate))
if theDate = "  /  /  "
  * invalid entry
else
  * valid entry
endif
```

Another common problem is with ranges of dates. It is not uncommon to reverse starting and ending dates, especially if the user is accustomed to thinking about the ending date as being more significant than the starting date. You should compare the two dates and swap them if the starting date is later than the ending. If the user enters the dates backward, an operation such as the following will lead to unexpected results.

```
copy to temp for OrderDate >= startDate ;
          .and. OrderDate <= endDate
```

Less obvious and sometimes more difficult to detect are correctly formatted dates that are not reasonable or even possible. For example, suppose you are associating line item shipment dates with the original purchase order records. It is not possible to ship an item before it was even ordered, so you should not allow a shipment date that is earlier than the original order date. These types of errors are extremely difficult to track down after the fact, but very simple to detect during data entry. It is easy to transpose two numbers in a date and have it look reasonable but still be in error.

Another similar situation is when a shipment date is too old to be reasonable for the original order. This is harder to detect because it may be possible to have items shipped, for example, 90 days beyond the order date. You cannot flag the date as an error and refuse to allow it because on a small number of occasions it may be correct. The users will probably like to have a warning that the date is suspect. They can double-check the date and be certain about it. You could provide a user-definable "maximum days old" parameter and allow them to fine-tune it themselves over the life of the application. If the range is too short the warning message pops up too frequently and gets ignored. If the range is too long it will not catch enough problems. Allowing the user to change the parameter provides an acceptable alternative.

Here is one more date problem to be aware of: When entering original documents (like purchase orders) you should establish some kind of "safety" date which no document can precede. Allow the user to move it forward periodically. For example, suppose that you install an application for use in 1991. Data prior to January 1, 1991 will not be kept in the system. If you check to be sure that no dates are prior to this you will cut down on potential data entry errors. Not all dates will be subject to the "safety date" check; for example, activity in January may reference dates in previous months. The check should be performed only on significant dates.

Another safety date could be established for the maximum allowable date. If data is to be entered for a particular fiscal year you can catch accidental entries too far into the future. If you give the user (or a system support/maintenance person) the ability to move these dates as the year progresses, you can trap a large number of data entry errors.

A common data entry error is transposing the digits in the year, entering 19 when you meant 91, or fumbling around the keyboard and producing 01, 09, and so on. A good date checking routine should catch all of these errors and at the very least pop up a warning message. Later in this chapter we will present message display functions ideally suited to this task.

Numbers

As with dates, there are a wide variety of simple checks that can be performed to catch bad numeric data. Ask your users about the kinds of data they encounter. Is zero ever a valid entry? Can an amount ever be negative? Can the sum of a set of transactions ever exceed the base amount? Should the sum of the transactions always equal the base amount? Is there an upper limit that is unlikely to be exceeded?

There is a big difference between saying that a field must accommodate five digits and saying that the maximum entry is 99,999. Your users may tell you that the maximum reasonable entry is more like 20,000, and an entry in the 50,000 range or higher should be impossible. Depending on the application, establishing some kind of floor and ceiling may help cut down on data entry errors.

Character strings

The use of PICTURE statements is encouraged, because they are your first line of defense. If you want the entry to be upper case, it's much easier to use the "@!" picture than to convert entries with the UPPER() function. Avoid using long strings of template symbols in field pictures, especially if the entry will be stored in a database. Database field widths are changed more often than we like, and every time it happens there are bound to be PICTURE templates that use the "old" width.

Here is an example. The customer name field was sworn to be 20 characters long.

```
@ 10, 15 get custName picture "!!!!!!!!!!!!!!!!!!!!"
```

Only near the end of the project did the client start encountering the really long names, so they requested it be increased to 25 characters. A clever programmer would avoid getting bitten like this again and store the picture in a variable.

```
@ 10, 15 get custName picture (pictCust)
```

This is better because changes to the width of the field need only affect the single assignment to the pictCust variable. However, there's an even easier way to accomplish the same thing.

```
@ 10, 15 get custName picture "@!"
```

Any picture template symbol can be used in this way. The picture will be as wide as the data it is associated with. It will vary on the fly, expanding and contracting to fit the data.

Screen layout

Screen layouts can cause problems at two different times: (a) when they are used at run-time; and (b) when they are originally programmed. Each problem makes the other one worse. Complex data entry screens are difficult to program, and screens that are difficult to program often look bad on the screen. This is a no-win situation.

Screen painting and code generating utilities are often a quick solution. However, they can sometimes create problems of their own by encouraging excessive use of boxes, colors, and other fancy tricks, simply because the tools are available and easy to use. Moreover, there is no time involved for laboriously typing all the complex formatting commands by hand.

This is not meant to discourage the use of screen painting and code generators; in fact, we think that with a well-designed template, you can (and usually will) generate well-designed applications. Just be sure that your templates are well-planned, and that the screens you paint with such utilities do not rely too heavily on gimmicks. A flashy, complicated screen is not a substitute for good design, and anyone with programming experience will be able to tell the difference.

Attempt to keep lines displayed on the screen to under 50 characters, and to write in complete sentences without abbreviations. Long (70+) character lines are difficult to read, especially if they wrap over several lines. If you liberally sprinkle the text with acronyms and abbreviations, the information quickly becomes unreadable, and consequently ignored.

One rule of thumb that you may wish to consider is: If you can't explain a prompt in a few short lines in a "how to" area at the bottom of the screen, then the prompt is too complicated and should be broken down. In such cases you could establish a direction (either the user wants to exit or continue) and then ask for additional information specific to the direction. In situations where there are several equally likely alternatives, a general-purpose menu function can be put to good use.

The natural way to read anything is from top to bottom and left to right, so this is how you should design your screens. Think of the screen as being divided into three zones (as shown in Figure 13.6). The topmost three lines or so almost never change. They are the "anchor", a point of reference. They establish the name of the application and the sub-module that is currently in use. This is desirable because, after all, your application is rarely the only one that runs on that particular computer. Even if it is, you should still indicate that your application is running.

Very inexperienced users often cannot distinguish between DOS and an application program, despite the seemingly obvious changes in screen formats. It is never acceptable to clear the entire screen and flash a "wait" message. The screen should always tell the user which application is making them wait, and even better, which process within the application is doing the work.

The middle zone is the obvious choice for messages and pertinent data, because the user's eyes will naturally be drawn to the center of the screen. This zone changes constantly as the user moves from prompt to prompt. It is cleared during long waits when the information on the screen is not applicable to the wait.

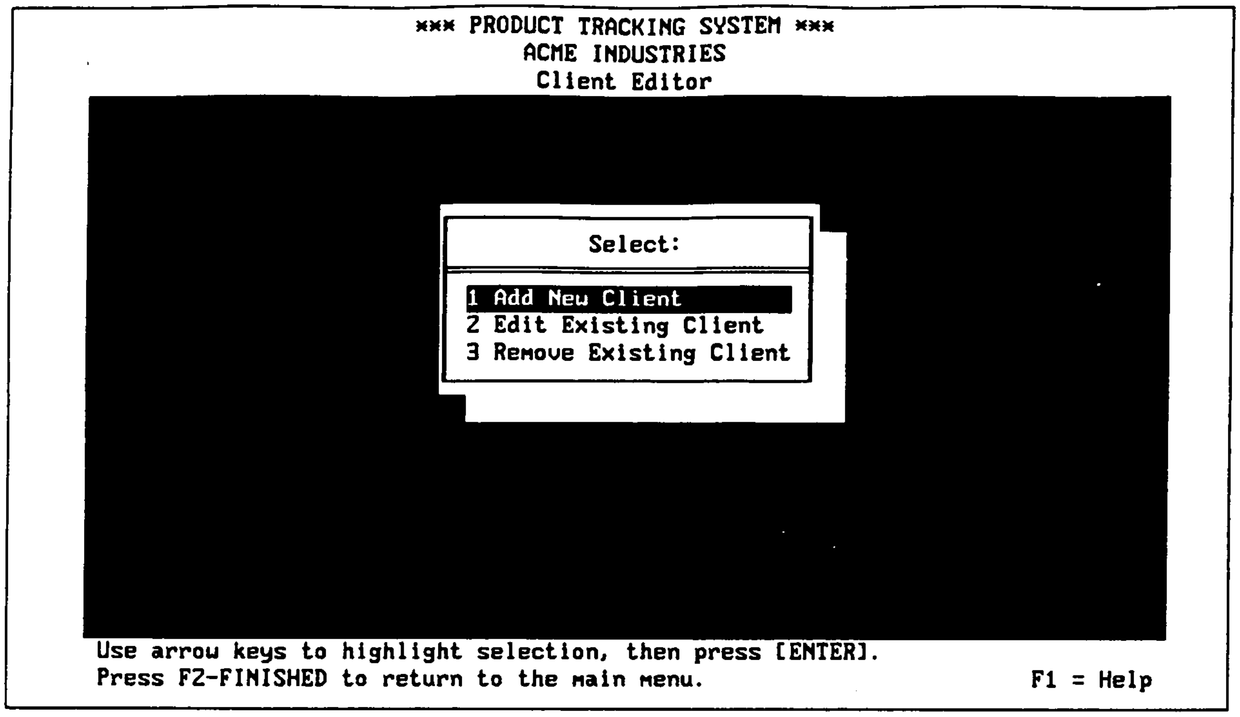
The bottom zone is for the instructional ("how to do it") text. It, too, changes constantly as the user moves from prompt to prompt. The changes are still uniform and consistent despite the difference in content. The "how to" text always pushes the text to the very bottom of the screen and separates itself from the middle zone of the screen, either with a line or a change in color.

The three-zone technique allows the user to concentrate on the middle portion of the screen, venturing to the top and bottom only when an unusual or unexpected situation arises.

The keyboard

Thus far we have talked only about displaying things on the screen. An equally important part of user interface design is the way you accept keystrokes from the end-user.

Figure 13.6 The Screen as Three Zones



```
*** PRODUCT TRACKING SYSTEM ***
ACME INDUSTRIES
Client Editor

Select:

1 Add New Client
2 Edit Existing Client
3 Remove Existing Client

Use arrow keys to highlight selection, then press [ENTER].
Press F2-FINISHED to return to the main menu.             F1 = Help
```

There are two major ways to handle the keyboard. One is to assign a number of keys to a number of functions and make them active almost all of the time. The "many keys" method is used by word processors and similar programs. At any given instant a huge number of keyboard combinations are available to perform different functions.

The other method is to activate a small number of keys and use them to invoke menus where the real functions are selected. Other programs have a relatively small number of active keys and use menu choices to perform functions. Most spreadsheets fall into this category. You hit a designated menu key and then start making selections from the menus.

A word processor which forces you to go through numerous menus is difficult to use, and similarly, in a complex application like a spreadsheet it is difficult to remember the hundreds of key combinations necessary to perform all the functions. Different applications require different approaches to the keyboard. Through trial and error (with heavy emphasis on the latter), we tend to use the "designated menu key" approach. Such applications are easier to use for beginning operators, and if implemented correctly, can also be efficient even for experienced users. Some programmers like to assign major functions to the ten function keys. They display a function key map at the bottom of the screen, something like the following.

```
F1      F2        F3       F4       F5      F6             F10

HELP    SEARCH    APPEND   DELETE   NEXT    PREVIOUS ...   EXIT
```

Such a technique is appropriate when rapid access to the various functions is important. However, problems begin to surface when the function key assignments are not maintained exactly the same way through the entire application. If function key F3 appends a new record on one data entry screen, it should append a record on *every* data entry screen, or be disabled for that screen. Applications which assign different functions to keys depending on the screen are difficult to use efficiently, because the user can never get comfortable with the keys. They will always have to stop to check the meaning of each key on each screen, which will lead to inefficiency and general bad feelings about your application.

When you start using SHIFT and CONTROL combinations to support all your features, you are getting further and further away from a balanced approach to the keyboard. There are a small number of applications that can benefit from dozens of special purpose keystrokes, but the majority will be better served by a menu-driven approach.

Once again, our approach follows our opinion of what makes a good user interface. After trying most methods, we keep coming back to the "designated menu key" concept with a few special purpose keys. This design is easier to implement, easier to explain to the user, easier to document, and easier to modify at a later date. Most of the time, the options that are possible apply only when the user has completed the current action. For example, during a data entry screen the append, next, previous, and search operations can only be used when the current record is no longer being edited.

Think of a data entry screen containing a glaring error that cannot be ignored. If the user presses any of the function keys (append a new record, move to next, previous and so on), you would have to alert them about the error and force them to either fix the current problem or discard the edits. Therefore, having all the functions available all of the time is not necessarily going to make things easier. If a single key is used to indicate "I'm done with this operation," you can do the error checking and loop back in a more natural way — "No, fix your errors and then you can leave". Once the screen is correct, the other options are displayed. If the option menu is laid out efficiently, you are adding a single extra keystroke to the call to any function. But two keystrokes can actually be faster than one if the single keystrokes are messy and inconsistent.

Using sound

The Summer '87 version of Clipper introduced the TONE() function. TONE() allows you to manipulate the IBM speaker to play tones at specified frequencies for varying durations. It is entirely possible that many Clipper developers have dismissed TONE() as being silly or unnecessary. If you count yourself among this group, we urge you to think again about using TONE().

Sound is an important part of any professional software package. It can be used in any of the following capacities:

- To alert the user that a time-consuming procedure is completed (e.g., reindexing all files, printing a complex report). Thus the user need not remain glued to a monitor; he or she can work on something else until the theme is heard.

- In conjunction with error messages. This is particularly useful for data entry-intensive applications where the user may be watching his or her data instead of the input screen.

- In tandem with alerts or warnings, (e.g., the user tries to print a report and the printer is off-line; the user is about to delete a large number of records, etc).

- To indicate different modes or states, if your application utilizes them. The user can associate each theme with a particular mode, which will prevent confusion.

- With data as a stimulus. There is at least one commercially available software package that displays data, and plays music corresponding to the patterns thereof. For example, where the numbers increase in value, the program might play a musical progression that began with low notes and went higher. (This type of integration of music and data is very much in its infancy.)

TONE()

The syntax for the TONE() function is:

```
TONE(<frequency>, <duration>)
```

<frequency> is an integer numeric expression representing the frequency (in hertz) at which to play the speaker.

<duration> is an integer numeric expression representing the duration of the tone in eighteenth of a second intervals. For example, passing a value of 18 would sound a tone for one second. If you do not pass this parameter, the default duration is 1/18 of a second.

The TONE() function does not return a value.

Examples

Near the end of Chapter 7, "The Preprocessor," we introduced several user-defined commands that played short musical themes. We will revisit two of those in Listing 13.4.

Listing 13.4 Charge and NannyBoo musical themes

```
#xcommand Charge => ;
              tunes({ {523,2}, {698,2}, {880,2}, ;
                    {1046,4}, {880,2}, {1046,8} } )

#xcommand NannyBoo => ;
              tunes( { {196,2}, {196,2}, {164,2}, ;
                    {220,2}, {196,4}, {164,4} } )

#xtranslate tunes(<a>) => ;
              aeval(<a>, { | a | tone(a\[1], a\[2]) } )
```

The "Charge" theme is handy for multi-user applications. If users are waiting for a file or record to become available, they can work on something else until they hear "Charge!", at which time they know they can return to their terminal.

"NannyBoo" is well suited to accompany error messages.

In addition, the following simple two-note combinations can be used to accompany alert or warning messages.

```
* Example 1

TONE(440,1)
TONE(440,1)

* Example 2
TONE(880,1)
TONE(880,1)
```

If you make use of sound in your applications, you should probably also consider allowing the user to change a global SOUND parameter to mute the application. There will always be some fuddy-duddies out there who think that any PC application that emits any kind of sound can only be a game.

Using color

Color is an extremely subjective thing, and some monitors do a better job than others with different color combinations. Everyone has seen software that comes with incredibly bad color combinations (which are more often than not hard-wired). Many are almost unreadable on laptop computer monitors because of the way laptops fake color through gray scales. Even on color monitors the color schemes are sometimes ludicrous, but somewhere out there a (color-blind?) programmer is mighty proud of his innovative use of color.

In your source code it is much easier to understand the intent of a reference to blinking or inverse text than it is to a particular color. There is a danger in using too many different color combinations simultaneously. Excessive colors obscure the significance of any given color. Flashy, colorful applications that look nifty during a demo might be too difficult to follow in actual use. Another potential problem is that the more colors you use, the better your chance for hideous color clashes.

Remember the FOCUS rule — a well-designed screen should be understood at a glance. If the user gets a long phone call right in the middle of your application and comes back 30 minutes later, will he be able to figure out where he was and what is expected of him next? Consistent and thoughtful use of color will make the screen easier to understand.

Avoid using hard-coded colors in your applications. Instead, refer to the situation that demands the use of some special handling. References to color mean nothing on a monochrome monitor (and in fact may not work at all they way you intend). Additionally, it is preferable to allow the users to modify (or at the very least have you modify for them) the colors used in their application. A comprehensive color selection and installation tool follows.

Color management

In Chapter 12 ("Program Design"), you tasted the power of file-wide static arrays. Now we will apply static arrays to the management of color settings. These principles can easily be applied to any type of global variables, including printer escape sequences, company/user identification, and so on.

Summer '87

If you used the Summer '87 version of Clipper, you might have handled your color settings by declaring **public** variables at the top of your main program, as shown in Listing 13.5.

Listing 13.5 Clipper Summer '87 Color Settings

```
* MAIN.PRG
public c_normal, c_bold, c_enhanced, c_blink, c_msg, c_warning
if iscolor()
   c_normal       := 'W/B'
   c_bold         := '+W/B'
   c_enhanced     := '+GR/B'
   c_blink        := '*W/B'
   c_msg          := '+W/RB'
   c_warning      := '+W/R'
else
   c_normal       := 'W/N'
   c_bold         := '+W/N'
   c_enhanced     := '+W/N'
   c_blink        := '*W/N'
   c_msg          := 'N/W'
   c_warning      := 'N/W'
endif
```

Another alternative was to put this code in a ColorInit() function that would in turn be called upon entering the main program. In either event, the variables would be declared as **public** so as to maintain visibility everywhere throughout the program.

When it came time to change color, the developer could pass one of these **public** variables to SETCOLOR() as shown in Listing 13.6.

Listing 13.6 Passing color variables to SETCOLOR()

```
* note: this is Summer '87 code!
whoops("File not available")
*
function whoops
parameter msg
private buffer, leftcol, oldcolor, oldrow, oldcol
oldrow = row()
oldcol = col()
leftcol = int( (76 - len(msg)) / 2)
```

```
oldcolor = setcolor(c_warning)      && PUBLIC color setting
buffer = savescreen(11, leftcol, 13, 80 - leftcol)
@ 11, leftcol, 13, 80 - leftcol box "┌─┐│┘─└│ "
@ 12, leftcol + 2 say msg
inkey(0)
** restore screen, cursor position, and color
restscreen(11, leftcol, 13, 80 - leftcol)
@ oldrow, oldcol say ''
setcolor(oldcolor)
return .t.
```

Clipper 5

With the tools that we had at our disposal at that time, the **public** declarations were the best solution. However, with the availability of the **static** declaration, there is no reason to use **public** variables in this fashion.

This is not meant to imply that **public** variables should be jettisoned altogether. There will continue to be uses in Clipper 5 for the **public** declaration, particularly if you plan to convert any major Summer '87 applications to Clipper 5. It would probably be too time-consuming for you to switch any such entrenched **public** color management routines in favor of the methods shown here.

However, as you write new applications in Clipper 5, you should strongly consider abandoning the use of **public** variables to manage your color (and other global) settings. The **static** declaration reveals two inherent deficiencies of **public** variables:

- As mentioned in Chapter 6 ("Variable Scoping"), public variables require an entry in the symbol table. This means that your programs will require more memory and execute more slowly. Static variables are faster than public variables because static (and local) variables are resolved at compile-time and thus do not require a symbol table entry.

Whenever your code refers to a **private** or **public** variable, the run-time engine must first look in the symbol table to determine that variable's actual address in memory. It then looks at that memory address to retrieve the value of the variable. Because static and local variables are already resolved, your program need only look in one place to retrieve their value.

Therefore, each time you refer to a public or private variable you are suffering a performance hit. Granted, this still happens in the blink of an eye, but if you multiply this by the thousands of times you refer to such variables, you can see the advantages of switching to static and local variables.

- Because public variables are visible everywhere throughout your program, that also means that they are subject to change everywhere. Computers are perfect, but unfortunately the people who program them are imperfect. These hapless humans occasionally make silly mistakes (especially when facing a deadline at three o'clock in the morning).

  There is nothing to prevent you from inadvertently trashing the value of a public variable, especially if you are not using a good naming convention for your variables. By contrast, static (and local) variables are visible only within the function or procedure that created them. This means that you will be unable to accidentally change them unless you are within the function that created them.

Encapsulation

The trick, therefore, is to tuck our color variables safely out of harm's way. The best approach is to "encapsulate" the variables (or data) along with the only function(s) that need to access them. Listing 13.7 shows a simple function, C_Normal(), that demonstrates encapsulation, along with a twist.

Listing 13.7 Simple encapsulation

```
function C_Normal(newcolor)
static color := "W/B"
/* change color setting if parameter was received */
if newcolor != NIL
   color := newcolor
endif
return setcolor(color)
```

C_Normal() declares a static variable **color** to white on blue. The function accepts an optional parameter, **newcolor**. If this parameter is passed, the static variable **color** will be assigned its value. You can see now why we must declare **color** as static rather than local, because we want it to retain its value for the next time we call C_Normal(). Local variables have to "start over" each time the function is called.

Finally, C_Normal() calls SETCOLOR() to actually change the color setting. It returns the SETCOLOR() return value, namely the current color setting, so that you can save this to a variable and restore it elsewhere.

There are two ways for you to utilize C_Normal(). The first is to call it normally:

```
C_Normal()
```

In this case it will simply change the color to white on blue (assuming that we had not yet changed the default color). However, you can also change this default by passing a parameter as shown in the following code fragment:

```
C_Normal("+w/r")
cls
setcolor("w/b")
? "this will be white on blue"
C_Normal()
? "this will be hi white on red"
```

This will cause the default normal color to be changed to bright white on red. Because the variable holding that value is static, it will retain its value for any subsequent calls to C_Normal(), as this example shows.

This effectively hides the normal color setting in the only function that needs to know it! You can no longer change it by accident. You can still change it if necessary, but because you have to make a conscious effort to pass the parameter, any such changes will be far more controlled than in the past.

If you look at this approach and think "Ugh... extra work", you're half right. It *is* going to require some additional thought when you write your code. But remember that far more time is spent during the maintenance (a.k.a. "debugging") phase than the coding phase. If you adhere to this approach, you will save yourself much time and many debugging headaches in the long run. You also make it much easier to change colors by keeping everything in just one place.

Function of many colors

C_Normal() was merely intended to demonstrate the encapsulation principle. In actual practice, there is little point in having one function for each color. Take the Clipper SET() function as an example. Clipper uses this function to handle all of the global settings (38 as of the release of Clipper 5.01; see Chapter 17, table 17.3, for a complete list of the SET manifest constants). The syntax for SET() is:

```
SET(<setting> [,<newvalue>])
```

The first parameter is a numeric expression that denotes which SETTING to look at. A complete list of manifest constants for each SETTING can be found in the SET.CH header file. For example, the CONSOLE setting can be referred to with this manifest constant:

```
#define _SET_CONSOLE          17

oldcons := set(_SET_CONSOLE, .F.)    // set console off
*
set(_SET_CONSOLE, oldcons)        // restore previous value
```

(Once again, you should always refer to the SETtings by their manifest constants rather than their numeric values. Nantucket warns that the numerics are subject to change, whereas the manifest constants will be maintained. Besides, the manifest constants are a lot easier to remember!)

Like C_Normal(), SET() accepts an optional **<newvalue>** parameter. If this parameter is passed, the global SETting is changed to the <newvalue>.

Inspired by this brevity, let's rewrite C_Normal() to handle all of our colors instead of just one (as shown in Listing 13.8). The gist of this change is to use a static array containing multiple color settings, which is easily accomplished.

Listing 13.8 ColorSet()

```
function ColorSet(colornum, newcolor)
static colors := { "W/B", "+W/B", "+GR/B", ;
                   "*W/B", "+W/RB", "+W/R" }
/* change applicable color setting if second parameter was passed
*/
if newcolor != NIL
   colors[colornum] := newcolor
endif
return setcolor(colors[colornum])
```

Let's also establish manifest constants so that we do not have to remember which array element corresponds to which color. Listing 13.9 shows these manifest constants, which will be stored in the COLOR.CH header file.

Listing 13.9 COLOR.CH: Manifest constants for color settings

```
/* COLOR.CH Header File */

#define  C_NORMAL     1
#define  C_BOLD       2
#define  C_ENHANCED   3
#define  C_BLINK      4
#define  C_MESSAGE    5
#define  C_WARNING    6
```

The Missing() function (shown in Listing 13.10) shows how you could refer to these colors.

Listing 13.10 Use of ColorSet() and Manifest Constants

```
#include "box. ch"
function Missing()
local oldcolor := ColorSet(C_BOLD)
local oldscrn := savescreen(11,30,13,49)
@ 11, 30, 13, 49 box B_SINGLE + " "
ColorSet(C_BLINK)
@ 12, 32 say "Record not found"
inkey(0)
setcolor(oldcolor)
restscreen(11, 30, 13, 49, oldscrn)
return nil
```

The Great Debate: color vs. monochrome

Actually, the answer is obvious. If God had intended for the world to be black and white, he would not have created rainbows. To our continuing dismay, even in the dawning of the age of VGA there are still numerous monochrome monitors in use. Therefore, we must devise an easy way to handle these aberrations.

We will have to follow two steps:

1. write a miniature function to initialize our color settings to either color or monochrome; and

2. expand the static array of color settings to include monochrome equivalents.

For the first step, we must declare a file-wide static variable **iscolor**. This is necessary because it will need to be visible in two functions.

The function ColorInit() will have one small but vitally important mission in life: to initialize **iscolor** to either true (yes, we are using color) or false (monochrome, yuch). If you call it without parameters, it will use the Clipper ISCOLOR() function as the basis for future colors.

However, as we have all learned by now, ISCOLOR() is not infallible. For example, ISCOLOR() will look at the everyday occurrence of a CGA video card and a monochrome monitor and return a value of true, even though the monitor obviously cannot display colors. Therefore, ColorInit() is structured to accept an optional parameter. If passed, this logical parameter will override ISCOLOR() to make the color vs. monochrome determination. Pass true for color, false for monochrome.

The second step is to modify the array in ColorSet() to hold monochrome settings for each color. This array must therefore be changed from a single-dimensional array of six elements, to an array containing six sub-arrays, each containing two elements. The first element of each sub-array will contain the color setting, and the second element will contain the monochrome setting. The static variable **iscolor** will serve as a pointer into the sub-array. See Listing 13.11 for ColorInit() and ColorSet().

Listing 13.11 ColorInit() and ColorSet()

```
static iscolor := 1

/* ColorInit(): initialize color system to color or mono */
function ColorInit(override)
iscolor := if(override == NIL, if(iscolor(), 1, 2), ;
              if(override, 1, 2))
return nil

/* ColorSet(): change colors in accordance with internal settings
*/
function ColorSet(colornum, newcolor)
static colors := {  { "W/B",    "W/N"  } , ;
                    { "+W/B",   "+W/N" } , ;
                    { "+GR/B",  "+W/N" } , ;
                    { "*W/B",   "*W/N" } , ;
                    { "+W/RB",  "N/W"  } , ;
                    { "+W/R",   "N/W"  } }

/* change color setting if second parameter was passed */
if newcolor != NIL
   colors[colornum, iscolor] := newcolor
endif
return setcolor(colors[colornum, iscolor])
```

The sample code shown in Listing 13.12 demonstrates how you can use ColorInit() and ColorSet() in your program.

Listing 13.12 Using ColorInit() and ColorSet()

```
function Main
/* verify that this is REALLY a color system */
if iscolor()
   ? "Press C for color monitor, any other key for monochrome"
   /* this looks nuts, but works perfectly - can you see why? */
   ColorInit( chr(inkey(0)) $ "cC" )
else
```

```
   ColorInit()
endif
ColorSet(C_NORMAL)
cls
ColorSet(C_BOLD)
@ 11, 24, 13, 55 box"┌─┐│┘─└│ "
ColorSet(C_ENHANCED)
@ 12, 26 say "Welcome to the Brownout Zone"
inkey(0)
return nil
```

Allowing changes

The final piece of this puzzle involves saving the color settings if they are modified. Clipper does not inherently provide for saving and restoring arrays. However, in Chapter 9 we presented two functions, SaveArray() and RestArray(), that accomplish this. We must rely upon these two functions to save and restore our color settings.

We also must make two slight changes to our functions. First, ColorInit() should be modified to accept a filename. If it receives one, it should be smart enough to attempt to load the color settings from that file.

Because the **colors** array must now be visible within ColorInit() as well as ColorSet(), we must pull it out of ColorSet() and make it file-wide (but you were expecting that anyway). To complete the scenario, we have provided the ColorMod() function, which displays samples of each color setting, and allows you to change them. You are then given the option of saving your color settings to a file.

Listing 13.13 shows ColorInit(), ColorSet() and ColorMod(), along with a test program demonstrating their use. Note that this relies upon the functions SaveEnv() and RestEnv(), which save and restore the environment. These were introduced in Chapter 12 ("Program Design"). Note that all of the color settings have been changed from the defaults to use cyan as the background.

Listing 13.13 Clipper 5 color management system

```
#include "box.ch"
#include "inkey.ch"
#define  TESTING     /* will compile stub program for testing */

#define  C_NORMAL    1
#define  C_BOLD      2
#define  C_ENHANCED  3
#define  C_BLINK     4
#define  C_MESSAGE   5
#define  C_WARNING   6
#define  COLOR_CNT   6

/* default name for color configuration file - change if you want
*/

#define  CFG_FILE    "colors.cfg"

// convert logical to numeric: 1 if .T., 2 if .F.
#translate Logic2Num( <a> ) => ( if( <a>, 1, 2) )

static iscolor :=1    //  flag for color (1) or mono (2)

/*
   The following array contains color and monochrome settings
   for each type of color. The third element describes the color
   it applies to, which makes it completely self-documenting. This
   third element is also used during the ColorMod() routine to
   identify each color.
*/

static colors := {  { "W/B",    "W/N" , "Normal"   }, ;
                    { "+W/B",  "+W/N", "Bold"      }, ;
                    { "+GR/B", "+W/N", "Enhanced" }, ;
                    { "*W/B",  "*W/N", "Blinking" }, ;
                    { "+W/RB", "N/W" , "Messages" }, ;
                    { "+W/R",  "N/W" , "Warnings" }  }

// stub program begins here
```

```
#ifdef TESTING

function Main
local oldcolor
do case
   case file(CFG_FILE)
       ColorInit(CFG_FILE)
   /* verify that this is REALLY a color system */
   case iscolor()
       qout("Press C for color monitor, any other key for mono")
       /* this looks nuts, but works perfectly - see why? */
       ColorInit( chr(inkey(0)) $ "cC" )
   otherwise
       ColorInit()
endcase
oldcolor := ColorSet(C_NORMAL)
cls
ColorSet(C_BOLD)
@ 11, 24, 13, 55 box "┌─┐│┘─└│ "
ColorSet(C_ENHANCED)
@ 12, 26 say "Welcome to the Brownout Zone"
inkey(3)
ColorMod()
ColorSet(C_BLINK)
@ 12, 26 say "Hope you enjoyed your visit!"
inkey(3)
setcolor(oldcolor)
cls
return nil
#endif
// stub program ends here... main functions begin

/*

     ColorInit(): initializes color management system
                  to either color or monochrome, or
                  load previously saved color settings

*/
```

```
function ColorInit(override)
local temparray
do case
   case override == NIL
        iscolor := Logic2Num(iscolor())
   case valtype(override) == 'L'
        iscolor := Logic2Num(override)
   otherwise
        if file(override)
           if len( temparray := Gloadarray(override) ) = 0
             qout("Could not load colors from " + override)
             inkey(0)
           else
             colors := temparray
           endif
           iscolor := logic2num( iscolor() )
        endif
endcase
return NIL
* end function ColorInit()
*--------------------------------*
/*
     ColorSet(): changes color in accordance with
                 internal settings stored in array
*/

function ColorSet(colornum, newcolor)
/* modify color setting if second parameter was passed */
if newcolor != NIL
   colors[colornum, iscolor] := newcolor
endif
return setcolor(colors[colornum, iscolor])
* end function ColorSet()
*--------------------------------*

/*
   ColorMod() - View/Modify all global color settings

*/
function ColorMod()
```

```
local key := 0, newcolor, ntop, xx, getlist := {}, colorfile, ;
  oldscore := set(_SET_SCOREBOARD, .f.)  // shut off scoreboard
SaveEnv(.t.)      // save entire screen for later restoration
ColorSet(C_NORMAL)
ntop := ( maxrow() - COLOR_CNT ) / 2
@ ntop, 22, ntop + COLOR_CNT + 1, 57 box B_SINGLE + ' '
setpos(ntop, 0)
/* pad each color setting to 8 characters for data entry */
aeval(colors, { | a, b | ;
      colors[b, iscolor] := padr(colors[b, iscolor], 8) } )
for xx := 1 to COLOR_CNT
   @ row() + 1, 24 say colors[xx, 3] + " Color"
   ColorSet(xx)
   @ row(), 42 say "SAMPLE" get colors[xx, iscolor];
                            valid Redraw(ntop)
   ColorSet(C_NORMAL)
next
read

/* trim each color setting */
aeval(colors,{ | a, b | ;
      colors[b, iscolor] := trim(colors[b, iscolor]) } )
setpos(ntop + COLOR_CNT + 1, 24)
dispout("Press F10 to save these settings")
if inkey(0) == K_F10
   colorfile := padr(CFG_FILE, 12)
   ColorSet(C_MESSAGE)
   @ 11, 18, 13, 61 box B_DOUBLE + ' '
   @ 12, 20 say "Enter file name to save to:"
   @ 12, 48 get colorfile picture '@!'
   setcursor(1)
   read
   setcursor(0)
   if lastkey() != K_ESC .and. ! empty(colorfile)
       Gsavearray(colors, ltrim(trim(colorfile)))
   endif
endif
RestEnv()
set(_SET_SCOREBOARD, oldscore)
return NIL
```

```
* end function ColorMod()
*-----------------------------------*

/*
   Redraw() - redraw color samples after each GET
*/
static function redraw(ntop)
local oldcolor := ColorSet(row() - ntop)
@ row(), 42 say "SAMPLE"
setcolor(oldcolor)
return .t.

* end static function Redraw()
*-----------------------------------*
```

Primitives

In earlier versions of Clipper, it was easy to run into glitches involving output. The most frequent problem was having an @..SAY message sent to the printer when it was intended for the screen. (This occurred because Clipper blindly followed the DEVICE setting.)

Fortunately, Clipper 5 gives us a much finer degree of control with a handful of new output functions:

Function	*Description*
DEVOUT()	Write a value to the current device
DEVOUTPICT()	Write a value to the current device w/ PICTURE clause
DEVPOS()	Move the cursor or printhead to a new position dependent upon the current state of SET DEVICE
DISPBOX()	Display a box on the screen
DISPOUT()	Write a value to the display
MAXCOL()	Returns maximum column that can be displayed on screen

MAXROW()	Returns maximum row that can be displayed on screen
OUTERR()	Write a list of values to the standard error device
OUTSTD()	Write a list of values to the standard output device
QOUT()	Display a list of expressions to next screen row
QQOUT()	Display a list of expressions to current screen position
SCROLL()	Scroll region of screen up or down, or optionally clear region of screen
SETCURSOR()	Controls size and shape of the cursor
SETMODE()	Changes display mode
SETPOS()	Move the cursor to a new position
SETPRC()	Reset positions of printer row and column (not new for Clipper 5 but relevant nonetheless)

Positioning Functions

DEVPOS(), SETPOS(), and SETPRC() all move the cursor and/or printhead. DEVPOS() moves either the cursor or printhead based on the current DEVICE setting: if SCREEN, the cursor is moved; if PRINTER, the printhead is moved. By contrast, SETPOS() and SETPRC() apply only to the cursor and printhead, respectively, regardless of the current device setting.

All three of these functions accept two parameters: <row> and<col>. These are numeric expressions representing the target row and column to which the cursor/ printhead should be moved.

If the cursor is moved, the values of ROW() and COL() (functions that return the current cursor position) will be updated accordingly. In the same fashion, printhead movement will update the values of PROW() and PCOL().

All three of these functions always return NIL.

The code fragment shown in Listing 13.14 demonstrates the use of these three functions, as well as how they are affected (or unaffected) by the SET DEVICE command.

Listing 13.14 Clipper 5 Primitive Output Functions

```
set device to print
setpos(10, 20)          // moves cursor
devpos(1, 2)            // moves printhead because of DEVICE
set device to screen
devpos(8, 0)            // now moves cursor because of DEVICE
setprc(10, 10)          // moves printhead
```

DEVPOS() Notes:

- If DEVPOS() is asked to move the printhead to a row less than the current PROW(), it will force a page eject.

- If DEVPOS() is requested to move the printhead to a column less than the current PCOL(), it will issue a carriage return and the required number of spaces.

- If the printer is redirected to a file using the SET PRINTER command, DEVPOS() updates the file instead of the printer.

- DEVPOS() is used in tandem with DEVOUT() to enable the @..SAY command. (See below for a discussion of @..SAY.)

Output functions

The following functions handle actual output: DEVOUT(), DEVOUTPICT(), DISPOUT(), QOUT(), QQOUT(), OUTERR(), and OUTSTD(). These functions fall neatly into logical groupings, so we will look at them in that context.

DEVOUT(), DEVOUTPICT(), and DISPOUT()

These three functions output a value. All of them ignore the current SET CONSOLE status. DEVOUT() and DEVOUTPICT() direct their output to the current device, as determined by either the SET DEVICE command or the SET() function. By contrast, DISPOUT() always writes its output to the screen, no matter what device is current. Though subtle, this distinction makes a world of difference, as we shall see shortly.

All of these functions accept two parameters: (a) the value to be displayed; and (b) a character expression representing the color in which to display the value on the screen.

The color parameter was new with release 5.01. It is optional, and (naturally) is ignored when output is directed to the printer. Being able to specify the color directly will cut down on many lines of code. If you have ever written something like this:

```
oldcolor := setcolor("+W/R")
@ 12, 20 say "WARNING!"
setcolor(oldcolor)
```

you will now be able to write it like so:

```
@ 12, 20 say "WARNING!" color "+W/R"
```

Clipper will take care of setting the color back to its previous setting.

DEVOUTPICT() also accepts a third parameter, which is a character string representing the PICTURE clause to use for displaying the value.

All three of these functions return a value of NIL. The code sample shown in Listing 13.15 demonstrates their use, and their relationship to the current device setting.

Listing 13.15 DEVOUT(), DEVPICTOUT() and DISPOUT()

```
function Test
cstring = "Test message"
devout(cstring)           // goes to screen
devout(cstring, "W/R")    // goes to screen in white on red
set device to print
devout(cstring)           // goes to printer
devout(cstring, "W/R")    // goes to printer, color ignored
devoutpict(cstring,,"@!") // goes to printer, all upper-case
dispout(cstring)          // goes to screen
dispout(cstring, "W/B")   // goes to screen in white on blue
return nil
```

@..SAY — to get a better idea of how these functions are used, let's review the preprocessor translations for the @..SAY command (as seen in the STD.CH header file):

```
#command @ <row>, <col> SAY <xpr> [PICTURE <pic>] ;
                                  [COLOR <color>] ;
                => DevPos( <row>, <col> )          ;;
                DevOutPict( <xpr>, <pic> [, <color>] )

#command @ <row>, <col> SAY <xpr> [COLOR <color>]    ;
                => DevPos( <row>, <col> )          ;;
                   DevOut( <xpr> [, <color>] )
```

DEVPOS() positions the cursor or printhead at the desired location. DEVOUT() outputs the desired value to the screen or printer. (If you specify a PICTURE clause, DEVOUTPICT() is used instead of DEVOUT()).

Note that DEVOUT() and DEVOUTPICT() are dependent upon the current device status, which means that your message could get misdirected to the printer. However, it is easy to ensure that your @..SAY messages always go to the screen. You can use the preprocessor to create your own user-defined command, @..SSAY, which relies upon the screen-specific SETPOS() and DISPOUT() functions. @..SSAY is shown in Listing 13.16.

Listing 13.16 @..SSAY

```
#xcommand  @ <row>, <col> SSAY <xpr> => ;
           SetPos( <row>, <col> ) ; DispOut( <xpr> )

function Main
set device to print
@ 10,10 say 'this goes to the printer'
@ row(),col() ssay 'this always goes to the screen'
return nil
```

We highly recommend that you add the @..SSAY command to your header file for future use. We also recommend that you write your user feedback functions (error messages, etc.) to make use of this command. That way, you will always be assured of having your message on the screen rather than the printer.

QOUT() and QQOUT()

These functions are the equivalent of the ? and ?? commands. They both output a list of values to the screen. The only difference between them is that QOUT() spits out a carriage return/linefeed combination prior to the values, whereas QQOUT() outputs the values at the current position.

These functions accept a comma-delimited list of values to be displayed. These values can be of any type (excepting array and block), and you do not have to concern yourself with converting everything to character string. The following statement demonstrates this:

```
qout(date(), 5, “string”, .f.)   // perfectly valid
```

(Note that these functions will output a space between each value in your list.)

These functions are both device-dependent, and thus write to either the screen or printer. If you use them to write to the screen, the values of ROW() and COL() will be updated accordingly. If they are used to write to the printer, PROW() and PCOL() will be updated.

At first glance, it would appear that QOUT() and QQOUT() are unnecessary because of the ? and ?? commands (which by the way are now translated by the preprocessor into calls to these functions). However, QOUT() and QQOUT() do indeed have their place. One such instance would be displaying values within an expression. Suppose that you want to display all the elements of an array with AEVAL() (introduced in Chapter 8, “Code Blocks”). You cannot use ? or ?? within an expression because the preprocessor will be unable to translate them! Listing 13.17 illustrates this principle.

Listing 13.17 QOUT() vs. the ? command

```
function Test
local a := { "qout()", "vs.", "?", "which", "will", "work" }
aeval(a, { | ele | ? ele } )        // will not compile
aeval(a, { | ele | qout(ele) } )   // much better!
return nil
```

OUTERR() and OUTSTD()

These two functions output a list of values. OUTERR() directs its output to the standard error device (**stderr**), whereas OUTSTD() writes to the standard output device (**stdout**).

As with QOUT() and QQOUT(), these two functions accept a list of values to be displayed. These values can be of any type (excluding array and block).

OUTERR() and OUTSTD() both return NIL.

Output from both of these functions bypasses the Clipper console stream. This means that you have no control over placement — functions such as DEVPOS() and SETPOS() will have absolutely no bearing.

However, you may have programs that will not require full-screen output. In those instances you can use these functions to avoid loading the terminal output subsystem, which will in turn save you approximately 25K in your executable file. To ease this process, you may wish to use the redefinitions of the ? and ?? commands shown in Listing 13.18 (which use OUTSTD() instead of DEVOUT()).

Listing 13.18 ? and ?? commands using OUTSTD()

```
#command  ? [ <list,...> ]    => outstd(chr(13)+chr(10)) ; ;
                                      outstd( <list> )
#command  ?? [ <list,...> ]    =>  outstd( <list> )
```

(The Nantucket-supplied SIMPLEIO.CH header file contains similar definitions.)

Unlike the other Clipper output functions, you may use DOS redirection with OUTSTD(). However, OUTERR() bypasses such redirection as well. The sample program shown in Listing 13.19 uses many of the output functions. You can readily see the destination of each function's output.

Listing 13.19 Redirection examples

```
/* TEMP.PRG */

function Main
set device to printer
qout("*to screen")
dispout("*to screen")
devout("*to printer")
outstd("*redirection")
outerr("*to screen")
return nil

D:\>temp
*to screen*to screen*redirection*to screen

D:\>temp > output
*to screen*to screen*to screen

D:\>type output
*redirection
```

DISPBOX()

DISPBOX() displays a box at specified coordinates. You can optionally specify a color parameter. The syntax is as follows:

```
DISPBOX( <t>, <l>, <b>, <r> [, <boxtype> ] [, <cColor> ] )
```

<t>, <l>, <b>, <r> are the coordinates of the box.

Optional parameter **<boxtype>** may be either a numeric or a character expression. Numeric value 1 indicates a single-line box, and 2 indicates a double-line box. Neither of these boxes will be cleared; i.e., they are the functional equivalent of the dBASE @ r,c TO r1,c1.

If you specify a character string, it will be treated as per the @..BOX command.

Optional parameter **<cColor>** is a character expression indicating the color in which to display the box. If you do not pass this, the current standard color will be used.

Examples of its use are:

```
DISPBOX(0, 0, 5, 10, 1)                // single line
DISPBOX(10, 1, 13, 50, 2)              // double line
DISPBOX(10, 1, 13, 50, 2, 'W/R')       // double, white on red
DISPBOX(20, 20, 22, 40, B_SINGLE)      // single line
```

SCROLL()

SCROLL() is used either to scroll a portion of the screen up or down a specified number of rows, or to clear a portion (or all) of the screen. Its parameters are as follows:

```
SCROLL( <nTop>, <nLeft>, <nBottom>, <nRight>, <nRows>)
```

<nTop>, <nLeft>, <nBottom>, and **<nRight>** represent the box coordinates. <nRows> is the number of rows to scroll. Positive values are scrolled down, and negative values are scrolled up. If you pass zero for <nRows>, the designated screen region will be cleared.

With the Clipper 5.01 release, all SCROLL() parameters are optional. If you do not send any coordinate parameters, the coordinates of the full visible screen is assumed. If you skip the fifth parameter (representing the number of lines to scroll), SCROLL() assumes that you want to clear the area.

If you look at the STD.CH file, you will notice that SCROLL() replaces previous internal functions __Clear() and __AtClear(). The following are excerpts from STD.CH.

```
#command @ <r>, <c> => Scroll( <r>, <c>, <r> ) ; ;
                        SetPos( <r>, <c> )

#command CLS => Scroll() ; SetPos(0,0)

#command @ <t>, <l> CLEAR ; => Scroll( <t>, <l> ) ; ;
                               SetPos( <t>, <l> )
```

Miscellaneous functions

SETMODE()

SETMODE() allows us to change the video display mode. It accepts two numeric parameters, **<rows>** and **<columns>**, and attempts to switch to the appropriate mode. **Undocumented feature**: If you pass a zero as one of these parameters, SETMODE() will *not* change that item.

SETMODE() returns a logical value: true if the mode change was successful, or false if it failed. If you have an EGA or VGA adapter, you could switch modes with the function shown in Listing 13.21.

Note that in order for a mode to be considered valid, that mode must be understood and recognized by both your hardware and the video driver.

Listing 13.21 Changing video modes

```
#define  EGA      43
#define  VGA      50
#define  REG      25
#define  SINGLE   80
#define  DOUBLE   40
#define  NOCHANGE  0

function Main
local oldrows := maxrow() + 1, oldcols := maxcol() + 1
cls
ChangeMode(VGA, SINGLE)
ChangeMode(NOCHANGE, DOUBLE)
ChangeMode(REG, 0)
ChangeMode(oldrows, oldcols)
return nil

function ChangeMode(rows, cols)
local ret_val := setmode(rows, cols)
if ! ret_val
   ? ltrim(str(rows)) + " x " + ltrim(str(cols)) + ;
        " mode not available"
else
   @ maxrow() - 2, 0 say "MAXROW(): " +str(maxrow())
   @ maxrow() - 1, 0 say "MAXCOL(): " +str(maxcol())
   @ maxrow(), 0 say "Current mode: " +ltrim(str(maxrow()+1))+;
                     " x " + ltrim(str(maxcol()+1))
endif
inkey(0)
return ret_val
```

MAXROW() and MAXCOL()

These functions return the maximum row and column positions that can be displayed.

The typical return values for these functions are 24 and 79, because most of your work will be in text mode (25 rows x 80 columns). However, as you have just seen with SETMODE(), it is now possible to use different display modes in your Clipper programs.

Although it may be awhile before you are predominantly working in graphics mode, you should expect it to be a question of "WHEN", not "IF". Accordingly, to save yourself a lot of drudgery later, you should adopt defensive programming techniques now. MAXROW() and MAXCOL() make it easy to do so:

- Instead of saving and restoring a screen like this:

  ```
  oldscrn := savescreen(0, 0, 24, 79)
  *
  restscreen(0, 0, 24, 79, oldscrn)
  ```

  you should save and restore it like so:

  ```
  oldscrn := savescreen(0, 0, maxrow(), maxcol())
  *
  restscreen(0, 0, maxrow(), maxcol(), oldscrn)
  ```

- When centering text on the screen, use MAXCOL() + 1 as the width instead of 80, because you might not always be in 80-column mode. (See the examples below for a display mode-independent CENTER() preprocessor function.)

SETCURSOR()

This function is a godsend for manipulating the size of the cursor. Prior to Clipper 5, the only way to do so was via the extremely limited SET CURSOR command. This command offered only two settings: ON and OFF. It was impossible to save the status of the cursor for restoration (as you would with screen contents and/or current color setting) without resorting to C or Assembler.

Thankfully, SETCURSOR() returns the current setting of the cursor. You can then save this to a memory variable or array element, change the cursor size, and restore it to its previous state.

That reason alone is sufficient to abandon the SET CURSOR command. But wait... there's more! You have greater flexibility with the size and shape of the cursor. Table 13.2 shows all available cursor types, along with the applicable parameter and manifest constant.

Table 13.2 Cursor types

Manifest Constant (in SETCURS.CH)	*Number*	*Description*
SC_NONE	0	No cursor
SC_NORMAL	1	Normal (underline)
SC_INSERT	2	Lower half block
SC_SPECIAL1	3	Full block
SC_SPECIAL2	4	Upper half block

Once again, we highly recommend that you refer to the manifest constants rather than the numeric values, which are subject to change. The code shown in Listing 13.22 illustrates the use of SETCURSOR() and these manifest constants.

Listing 13.22 Changing the cursor with SETCURSOR()

```
#include "setcurs.ch"    // for using the manifest constants

function Whatever
local oldcursor := setcursor(SC_NONE)  // turn off cursor
*
*
setcursor(SC_SPECIAL1)         // full block cursor
read
setcursor(SC_NONE)             // turn off cursor again
*
*
setcursor(oldcursor)           // restore to previous state
return nil
```

Examples

Centering text

Centering text on the screen is a basic need for any Clipper application. Listing 13.23 presents two user-defined commands. CENTER() centers text on either the screen or printer, dependent upon the current device. SCRNCENTER() always centers the text on the screen, thanks to SETPOS() and DISPOUT().

Listing 13.23 Centering text on screen and printer

```
#xtranslate CENTER(<row>, <msg> [,<color>] ] => ;
   DevPos( <row>, int(( maxcol()+1 - len(<msg>)) / 2)) ; ;
   DevOut( <msg> [, <color>] )

#xtranslate SCRNCENTER(<row>, <msg> [, <color>] ) => ;
   SetPos( <row>, int(( maxcol()+1 - len(<msg>)) / 2)) ; ;
   DispOut( <msg> [, <color>] )

function test
set device to print
center(1, "this goes to the printer")
scrncenter(1, "this goes to the screen")
return nil
```

As mentioned previously, note that both CENTER() and SCRNCENTER() make use of the MAXCOL() function to determine the width for centering your message. This makes them oblivious to changes in video mode, thus saving you valuable time recoding your applications should you decide to change to a graphical user interface in the future.

Saving/restoring cursor position

If you wanted to save and restore the cursor position in prior versions of Clipper, you probably used logic similar to that shown in Listing 13.24.

Listing 13.24 Saving/restoring cursor position in Summer '87

```
function MyFunc
private mrow, mcol
mrow = row()
mcol = col()
*
@ mrow, mcol say ''
return ret_val
```

There are two reasons why you would not want to do this in Clipper 5:

- The preprocessor converts the @..SAY command into calls to DEVPOS() and DEVOUT(). Because you do not want to display a value, the DEVOUT() call is completely unnecessary.

- If the device is set to PRINT, the printhead position will be changed instead of the cursor, thus undermining your housekeeping attempts.

Therefore, you should use the following command to reposition the cursor:

```
setpos(mrow, mcol)
```

This kills both birds with one line by eliminating the useless function call and forcing the cursor to be updated regardless of the current device.

In similar fashion, there is no longer any need for you to write code such as the following:

```
@ prow(), 1 say cust->lname
@ prow(), pcol(), say cust->fname
```

Because you want to print **fname** at the current printhead position, you need only call DEVOUT(). Thus you will avoid a redundant DEVPOS() function call.

User feedback on large databases

Suppose that you want to sum a numeric field on a 50,000 record database. As you probably already know, the SUM operation does not give any user feedback whatsoever. Given a large enough database (and a slow enough CPU), a timid user might think that the machine is locked up, and thus be tempted to reboot. Heaven forbid!

The COUNT, SUM, and AVERAGE commands are translated by the preprocessor into DBEVAL() calls. As discussed in Chapter 8 ("Code Blocks"), DBEVAL() evaluates a code block for a given set of records in a database. It is therefore quite easy for us to put "hooks" into this code block that will display the current record number being processed, which will give the user a clear indication that the computer is indeed clicking away.

The following command:

```
sum ITEM->quantity to temp
```

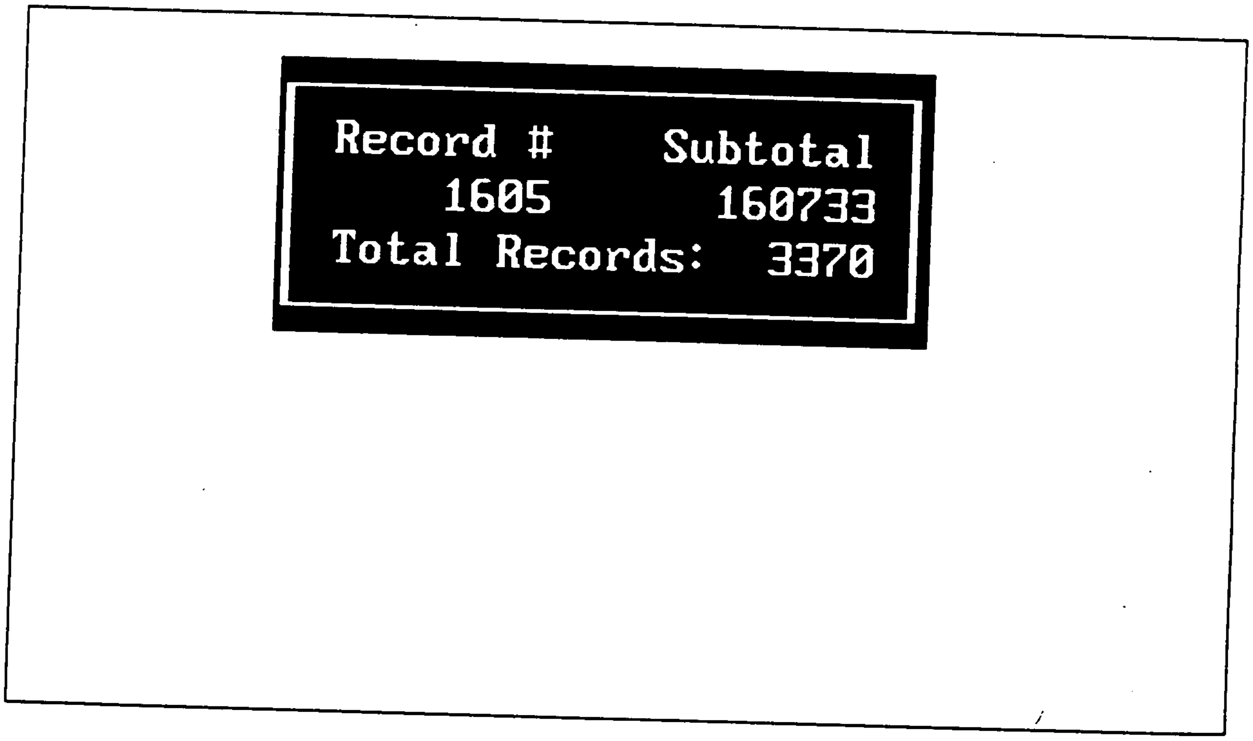
is translated to:

```
temp := 0 ; ;
DBEval( {|| temp := temp + item->quantity},,,,, .F.)
```

This is fairly straightforward. The variable **temp** is initialized to zero, and will be incremented by the value of **quantity** for each record in the database. (Note that you can easily limit the scope of this command by supplying one or more other code blocks to DBEVAL(). For more specifics, please refer back to Chapter 8.)

Two function calls are added to the code block to display the current record number and running total. SETPOS() positions the cursor where we want it, and QQOUT() displays the values. While we're at it, we will also use the compound assignment operator "+=", which should have been in there in the first place. Listing 13.25 shows the source code for this code block, and Figure 13.7 shows how it looks on the screen.

Figure 13.7 User feedback during SUM operation



```
Record #     Subtotal
   1605        160733
Total Records:   3370
```

Listing 13.25 User feedback during SUM operation

```
#include "box.ch"

function Test
cls
setcolor('+w/b')
use item
dispbox(10, 28, 14, 51, B_SINGLE + ' ')
@ 11, 30 say "Record #   Subtotal"
@ 13, 30 say "Total Records: " + ltrim(str(lastrec()))
temp := 0
DBEval( {|| temp += item->quantity, ;
        setpos(12, 30), qqout(recno(), temp) },,,,, .F. )
return nil
```

Displaying page numbers while printing

Listing 13.26 demonstrates the use of SETPOS() and DISPOUT() to display page numbers on the screen while printing a report. The use of these two functions precludes you from having to switch the DEVICE setting back and forth (which can get very confusing very quickly).

Listing 13.26 Displaying page numbers while printing

```
#include "box.ch"
function Report
use customer
dispbox(11, 28, 13, 52, B_SINGLE + ' ')
@ 12, 30 say "Now printing page "
set device to print
heading(init)
do while ! eof()
   @ prow()+1, 0 say trim(customer->lname) + ', ' + ;
                     left(trim(customer->fname), 1)
   @ prow(),  25 say customer->addr1
   @ prow(),  55 say customer->city
   @ prow(),  70 say customer->state
   @ prow(),  74 say left(customer->zip, 5)
```

```
      if prow() > 57
         heading()
      endif
      skip
   enddo
   set device to screen
   return nil

   function Heading(init)
   static page := 1
   /*
      if parameter was passed to Heading(), we are in initialize mode
      and must reset the page counter - this is necessary if you run
      this report a second (or third) time
   */
   if init != NIL
      page := 1
   endif
   @ 0, 0 say "Customer List - Page " + ltrim(str(page))
   setpos(12, 48)
   dispout(str(page++, 3))     // increment page counter
   return nil
```

The function Heading() declares **page** as a static variable, which means that it will retain its value each time you re-enter the function. However, this does have its drawbacks, which become painfully obvious when you run this report a second time and start at page 20! Therefore, the first time you call Heading(), you should pass it any parameter, which instructs the function to re-initialize the page counter to 1.

Heading() moves the printhead to the top of the page and displays a brief heading along with the page number. It then displays the page number on the screen in the message box, and increments the page counter.

User interface building blocks

Now that you understand the Clipper primitives, let's create a few specialized user interface tools that cut the task of user interface programming down to size.

That darn cursor

A very simple thing you can do to spruce up your Clipper screen displays is to turn the cursor off. The only time it needs to be on is during READ and MEMOEDIT(), and those are easy enough to track down. Simply turn the cursor on right before a READ or MEMOEDIT() and then off again when they are finished. If you write a function that needs to turn the cursor on, save the current state and be sure to restore it on exit from the function. All the functions in this section either assume the cursor is already off, or they turn it off and restore it on exit. Fortunately, this is easy to do, as Listing 13.27 demonstrates.

Listing 13.27 Managing the cursor

```
//  Save existing cursor state (could be on or off)
//  and then shut it off.
curs := setcursor(0)
//  Restore original cursor state.
setcursor(curs)
```

Most of the "SET" series of functions work like this. Frequently you'll need to save more than just the cursor. If you are going to mess up the screen and change colors, your functions should clean up after themselves and restore the entire environment. See Chapter 12, "Program Design," for some excellent functions that accomplish this task via the use of static arrays.

Yes/No questions

For the amount of information you ultimately receive from them, hard-wired yes/no prompts are more trouble than they're worth. Consider the amount of code they take to implement, as shown in Listing 13.28.

Reexamination 90/005,727

The Original

Page 574 of Part III

Is Missing

Initial : AR
Date : 6-8-2000

Listing 13.28 Hard-wired Yes/No prompt

```
@ 10, 15 say "Are you sure (Y/N)?"
k := inkey(0)
if upper(chr(k)) == "Y"
  zap
endif
```

No Clipper programmer should have to go through all that when user-defined functions are available. Listing 13.29 shows the Yes() function, which greatly simplifies the task.

Listing 13.29 Yes() function for Yes/No prompt

```
function Yes(r, c, message)
@ r, c say message
return (upper(inkey(0)) == "Y")
```

The previous example can now be reduced to the following:

```
if Yes(10, 15, "Are you sure (Y/N)?")
  zap
endif
```

While this alone is a substantial improvement, we are now free to add more features to the basic function. YesBox() (shown in Listing 13.30) makes the message more visible to the end-user by drawing a box around it.

Listing 13.30 YesBox()

```
#include "box.ch"

function YesBox(r, c, message)
//  Draw a box that will surround the message.
dispbox(r, c, r + 2, c + len(message), B_SINGLE + ' ')
```

```
// Display the message.
@ r + 1, c + 1 say message
return (upper(inkey(0)) == "Y")
```

Note how adding such features costs us nothing — the function call stays simple while the function itself can get as complex as we need to support the features. This is the basic concept behind user interface building blocks.

There are other features that can be added to illustrate more of these building block concepts. One such feature is a "time-out," where if the user does not press a key before a certain amount of time, "no" is assumed and the function continues. Listing 13.31 adds this feature to YesBox().

Listing 13.31 YesBox() with time-out

```
#include "box.ch"

function YesBox(r, c, message, wait)
// Draw a box that will surround the message.
dispbox(r, c, r + 2, c + len(message), B_SINGLE + ' ')
// Display the message.
@ r +1, c +1 say message
return (upper(inkey(wait)) == "Y")
```

While a time-out feature is very handy when you need it, it gets in the way when you do not want it. As written above, YesBox() requires a time period or it will crash when it hits the INKEY() function. It would be better to make the time period parameter optional. If it is not specified, the function should wait "forever" as did the earlier versions of the function. As you saw in Chapter 4 ("Data Types"), if you do not pass a parameter, Clipper assigns it a value of NIL, so the missing parameters are easy to test.

Chapter 7, "The Preprocessor", presented examples of this logic. This chapter also presented an incredibly handy user-defined command: DEFAULT. DEFAULT is interpreted as: Take a look at the value of **<a>**. If it's NIL, assign the default value of **<b>**; otherwise, leave it alone. Listing 13.32 presents another look at DEFAULT.

Listing 13.32 DEFAULT

```
#xcommand DEFAULT  <p> TO <v> [, <p2> TO <v2> ] => ;
                   <p> := IF(<p> == NIL, <v>, <p>);
                   [; <p2> := IF(<p2> == NIL, <v2>, <p2>) ]
```

DEFAULT can be used to assign the **wait** parameter a value of zero if it is not specified in the function call (see Listing 13.33).

Listing 13.33 YesBox() with DEFAULT

```
#include "box.ch"

function YesBox(r, c, message, wait)
default wait to 0
//  Draw a box that will surround the message.
dispbox(r, c, r + 2, c + len(message), B_SINGLE + ' ')
//  Display the message.
@ r +1, c +1 say message
return (upper(inkey(wait)) == "Y")
```

YesBox() can now be called with or without the time-out parameter, and it will react in a manner consistent with what you would expect from the syntax.

```
if YesBox("Take your time... Y or N?")
  if YesBox("Quick! You've got three seconds! Y/N?", 3)
  endif
endif
```

While you're at it, why not make the row and column parameters optional, too? All that is necessary are reasonable default values for each parameter. The most reasonable thing to do is center the message on the screen. If no row is specified, start the box in the middle. If no column is specified, center the message on the screen. (Remember that the DEFAULT #xcommand must be included in the source code file.) Listing 13.34 demonstrates these additions to YesBox(). Note the use of MAXROW() and MAXCOL() to ensure video-mode independence.

Listing 13.34 YesBox() with default row and column

```
#include "box.ch"

function YesBox(r, c, message, wait)
// assign default values
default r to (maxrow() / 2) - 1
default c to (maxcol() - len(message)) / 2
default wait to 0
//  Draw a box that will surround the message.
dispbox(r, c, r + 2, c + len(message), B_SINGLE + ' ')
//  Display the message.
@ r + 1, c + 1 say message
return (upper(inkey(wait)) == "Y")
```

At this point you have an extremely versatile user interface function capable of making screen position and duration assumptions based on how it is called, and you have not even worked up a sweat! Listing 13.35 shows how you could phrase your calls to YesBox() in its current configuration.

Listing 13.35 YesBox() sample calls

```
if YesBox(,, "Is this centered on the screen?")
  //  We wrote the function correctly.
endif

if .not. YesBox(17,, "Is this centered on row 17?")
  //  We screwed up somewhere – time to debug.
endif

if YesBox(, 20, "Is this starting in column 20?")
  //  Amazing, we can skip any combination.
endif
```

Let's pull out all the stops and write the ultimate in Yes/No prompts, one that uses a lightbar menu for a "dialog box" feel. You should also be able to specify a color to use. Of course, everything but the actual prompt message should be optional. Here's an example of a call that uses all the parameters.

```
answer := YesNoMenu(10, 15, "Are you sure?", C_MESSAGE)
```

This doesn't look much different than the YesBox() function. Things are starting to get more complex inside the function. However, the calls to the function remain easy to read and remember because you have laid out the parameters and given some thought to default values. Listing 13.36 contains the source code for YesNoMenu(). (Note that this function calls on the ColorSet() function, discussed previously in this chapter, for color handling.)

Listing 13.36 Lightbar menu-based Yes/No prompt

```
#include "box.ch"
#include "colors.ch"
function YesNoMenu(r, c, msg, clr)
/*
     General-purpose Yes/No prompt function.
     Parameter  Description

     r          Starting row, default is center of screen.
     c          Starting column, default is center of screen.
     msg        Message to display as a prompt.
     clr        Color number to use, default is current color.

     Note: Color selection for this function is handled by
     the ColorSet() function.

*/

local response                  //  User's response
local r2, c2, offset            //  Help with box coordinates
local pYes, pNo                 //  Prompts for "Yes" and "No"

//  Make sure all parameters have reasonable default values.
default r to (maxrow() / 2) -2
default c to (maxcol() - len(msg)) / 2
default clr to C_NORMAL
//  Prompts to display for choices.
pYes := "YES"
pNo  := "NO"
/*
     Calculate ending box coordinates. Box width must be able to
     accommodate the longest of either the prompt message or the
     YES and NO selections.
*/

r2 := r +3
c2 := c +len(msg) +1
c2 := max(c2, c +len(pYes) +2 +len(pNo))
```

```
// Calculate how far to indent Yes/No prompts within the box.
offset := ((c2 -c) -(len(pYes) +2 +len(pNo))) /2
// Save existing screen, color etc. (SaveEnv() is in Chapter 12)
SaveEnv()
// Clear the area and draw a box
dispbox(r, c, r2, c2, B_SINGLE + ' ')
// Display message and Yes/No prompts, wait for response.
@ r +1, c +1 say msg
@ r +2, c +1 +offset prompt pYes
@ r +2, col() +2  prompt pNo
menu to response
// Restore original screen, color etc before returning.
RestEnv()
return (response ==1)
```

A careful reading of the source code shows that YesNoMenu() takes great pains not to leave any "tracks" on the screen. Everything related to the screen is left as it was before the call to the function. YesNoMenu() can pop up over a complicated screen, ask its question, and disappear without disturbing anything. This is not simply a nice convenience feature — it is the only way that general-purpose functions should be written. The amount of effort you put into the functions is paid back many times in more reliable and consistent applications.

ALERT()

ALERT() is a function added with the 5.01 release of Clipper. Although it is used primarily in the run-time error handler (ERRORSYS.PRG), you will find plenty of situations in your programs (apart from error handling) where you will be able to make excellent use of ALERT().

It is a marvelous little function that displays a message, and presents one or more options for the user to select from. The syntax is:

```
ALERT( <cMessage> [, <aOptions> ][, <cColor> ] )
```

<cMessage> is the message to be displayed.

Optional parameter **<aOptions>** is an array of options. If you do not use this, the only option will be "OK".

Undocumented optional parameter <cColor> is the color in which to display the box. The default colors are hi white on red for the box and message, and hi white on blue for the highlighted option. Bear in mind that since this parameter is undocumented, it is subject to future change.

ALERT() displays a box centered on the screen. Your message is centered on the top row within the box, and the options are centered just below that.

ALERT() returns a numeric value representing the option that was selected. For example, if the user selected "Quit" in the following code fragment, ALERT() would return 2. As with MENU TO and ACHOICE(), if the user exits with Esc ALERT() will return a zero.

```
ALERT("Something horrible has happened", ;
      { "Ignore", "Quit" })
```

NOTE: ALERT() is sensitive to the presence (or absence) of the full-screen input/output subsystem. If your program does not make use of these functions, ALERT() will display its messages using output of the standard device, as opposed to full-screen output. This means that the pleasant lightbar interface will not be used, but the idea will still be clear enough to the user.